HI-TECH C® for PIC10/12/16
User's Guide

Note the following details of the code protection feature on Microchip devices:

- Microchip products meet the specification contained in their particular Microchip Data Sheet.

- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.

- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.

- Microchip is willing to work with the customer who is concerned about the integrity of their code.

- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as "unbreakable."

Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products. Attempts to break Microchip's code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.

Trademarks

QUALITY MANAGEMENT SYSTEM
CERTIFIED BY DNV
ISO/TS 16949:2002

Table of Contents

Chapter 1. HI-TECH C Compiler for PIC10/12/16 MCUs

1.1   OVERVIEW

This manual describes the usage and operation of the HI-TECH C Compiler for PIC10/12/16 MCUs.

The HI-TECH C Compiler for PIC10/12/16 MCUs is a free-standing, optimizing ANSI C compiler. It supports all PIC10, PIC12 and PIC16 series devices, as well as the PIC14000 device and the enhanced Mid-Range PIC® MCU architecture.

The compiler is available for several popular operating systems, including 32 and 64-bit Windows®, Linux and Apple OS X.

As well as being a stand-alone console application, it is fully compatible with Microchip's MPLAB IDE, allowing source-level debugging with the MPLAB ICE in-circuit emulator, the MPLAB ICD 2 in-circuit debugger or the MPLAB SIM simulator.

The compiler also integrates into HI-TIDE. This is an IDE based on Eclipse, and is available for Windows, Linux and Mac OS X platforms.

1.2   CONVENTIONS

Throughout this manual, the term "the compiler" is often used. It can refer to either all, or some subset of, the collection of applications that form the HI-TECH C Compiler for PIC10/12/16 MCUs. Often it is not important to know, for example, whether an action is performed by the parser or code generator application, and it is sufficient to say it was performed by "the compiler".

It is also reasonable for "the compiler" to refer to the command-line driver (or just driver) as this is the application that is always executed to invoke the compilation process. The driver for the HI-TECH C Compiler for PIC10/12/16 MCUs package is called PICC. The driver and its options are discussed in **Chapter 1. "HI-TECH C Compiler for PIC10/12/16 MCUs"**. Following this view, "compiler options" should be considered command-line driver options, unless otherwise specified in this manual.

Similarly "compilation" refers to all, or some part of, the steps involved in generating source code into an executable binary image.

NOTES:

Chapter 2. PICC Command-line Driver

2.1 OVERVIEW

The command-line driver is called PICC™ and is the application that can be invoked to perform all aspects of compilation, including C code generation, assembly and link steps. Even if you use an IDE to assist with compilation, the IDE will ultimately call PICC.

Although the compiler applications can be called explicitly from the command line, using PICC is the recommended way to use the compiler as it hides the complexity of all the internal applications used and provides a consistent interface for all compilation steps.

This chapter describes the steps the driver takes during compilation, files that the driver can accept and produce, as well as the command-line options that control the compiler's operation. It also shows the relationship between these command-line options and the controls in the MPLAB IDE *Build Options* dialog.

2.2 INVOKING THE COMPILER

This section looks at how to use PICC as well as the tasks that it, and the internal applications, perform during compilation.

PICC has the following basic command format.

```
PICC [options] files [libraries]
```

It is assumed in this manual that the compiler applications are either in the console's search path, or the full path is specified when executing any application. The compiler's location can be added to the search path when installing the compiler by selecting the *Add to environment* checkbox in the install program.

It is conventional to supply `options` (identified by a leading *dash* "-" or double *dash* "–") before the filenames, although this is not mandatory.

The formats of the options are discussed in **Section 2.7 "PICC Driver Option Descriptions"**, and a detailed description of each option follows.

The `files` may be any mixture of C and assembler source files, and precompiled intermediate files, such as relocatable object (`.obj`) files or p-code (`.p1`) files. The order of the files is not important, except that it may affect the order in which code or data appears in memory, and may affect the name of some of the output files.

`Libraries` is a list of used-defined object code or p-code library files that will be searched by the linker in addition to the standard C libraries. The order of these files will determine the order in which they are searched. They are typically placed after the source filename, but this is not mandatory.

PICC distinguishes source files, intermediate files and library files solely by the file type, or extension. Recognized file types are listed in Table 2-1. Alphabetic case of the extension is not important from the compiler's point of view, but most operating system shells are case sensitive.

TABLE 2-1: PICC™ INPUT FILE TYPES

File Type	Meaning
.c	C source file
.p1	p-code file
.lpp	p-code library file
.as or .asm	Assembler source file
.obj	Relocatable object code file
.lib	Relocatable object library file
.HEX	Intel HEX file

This means, for example, that a C source file must have a .c extension. Assembler files can use either .as or .asm extensions.

The terms "source file" and "module" are often used when talking about computer programs. They are often used interchangeably, but they refer to the source code at different points in the compilation sequence.

A source file is a file that contains all or part of a program. Source files are initially passed to the preprocessor by the driver. A module is the output of the preprocessor, for a given source file, after inclusion of any header files (or other source files) which are specified by #include preprocessor directives. These modules are then passed to the remainder of the compiler applications. Thus, a module may consist of several source and header files. A module is also often referred to as a translation unit. These terms can also be applied to assembly files, as they too can include other header and source files.

2.2.1 Output Files

There are many files created by the compiler during the compilation. A large number of these are intermediate files and are usually deleted after compilation is complete, but several remain and are used for programming the device, or for debugging purposes.

The main output file that will contain the machine code encoding of the original C program will default to a particular type, but this can be controlled by compiler options, e.g. the --OUTPUT option. The extensions used by these files are fixed and are listed together with this option's description in **Section 2.7.44 "--OUTPUT= type: Specify Output File Type"**.

The names of many output files use the same base name as the source file from which they were derived. For example the source file input.c will create a p-code file called input.p1. However some of the output files contain project-wide information and are not directly associated with any one particular input file, e.g. the map file. If the names of these output files are not specified by a compiler option, their base name is derived from the first C source file listed on the command line. If there are no files of this type specified, the name is based on the first input file (regardless of type) on the command line.

If you are using an IDE, such as MPLAB® IDE, to specify options to the compiler, there is typically a project file that is created for each application. The name of this project is used as the base name for project-wide output files, unless otherwise specified by the user. However check the manual for the IDE you are using for more details.

> **Note:** Throughout this manual, the term *project name* will refer to either the name of the project created in the IDE, or the base name (file name without extension) of the first C source file specified on the command line.

2.2.2   Long Command Lines

The PICC driver is capable of processing command lines exceeding any operating system limitation. To do this, the driver may be passed options via a command file. The command file is specified by using the @ symbol which should be immediately followed (i.e. no intermediate space character) by the name of the file containing the command line arguments intended for the driver.

Each command-line argument must be separated by one or more spaces and may be placed over several lines by using a space and *backslash* character to separate lines. The file may contain blank lines, which are simply skipped by the driver.

The use of a command file means that compiler options and project filenames can be stored along with the project, making them more easily accessible and permanently recorded for future use., but without involving the complexity of creating a make utility.

For example a command file `xyz.cmd` is constructed any text editor and contains both the options and file names that are required to compile your project as follows.

```
--chip=16F877A -m \
--opt=all -g \
main.c isr.c
```

After it is saved, the compiler may be invoked with the command:

```
PICC @xyz.cmd
```

2.3   THE COMPILATION SEQUENCE

The main compiler applications and files are illustrated in Figure 2-2.

FIGURE 2-1: **COMPILER APPLICATIONS AND FILES**
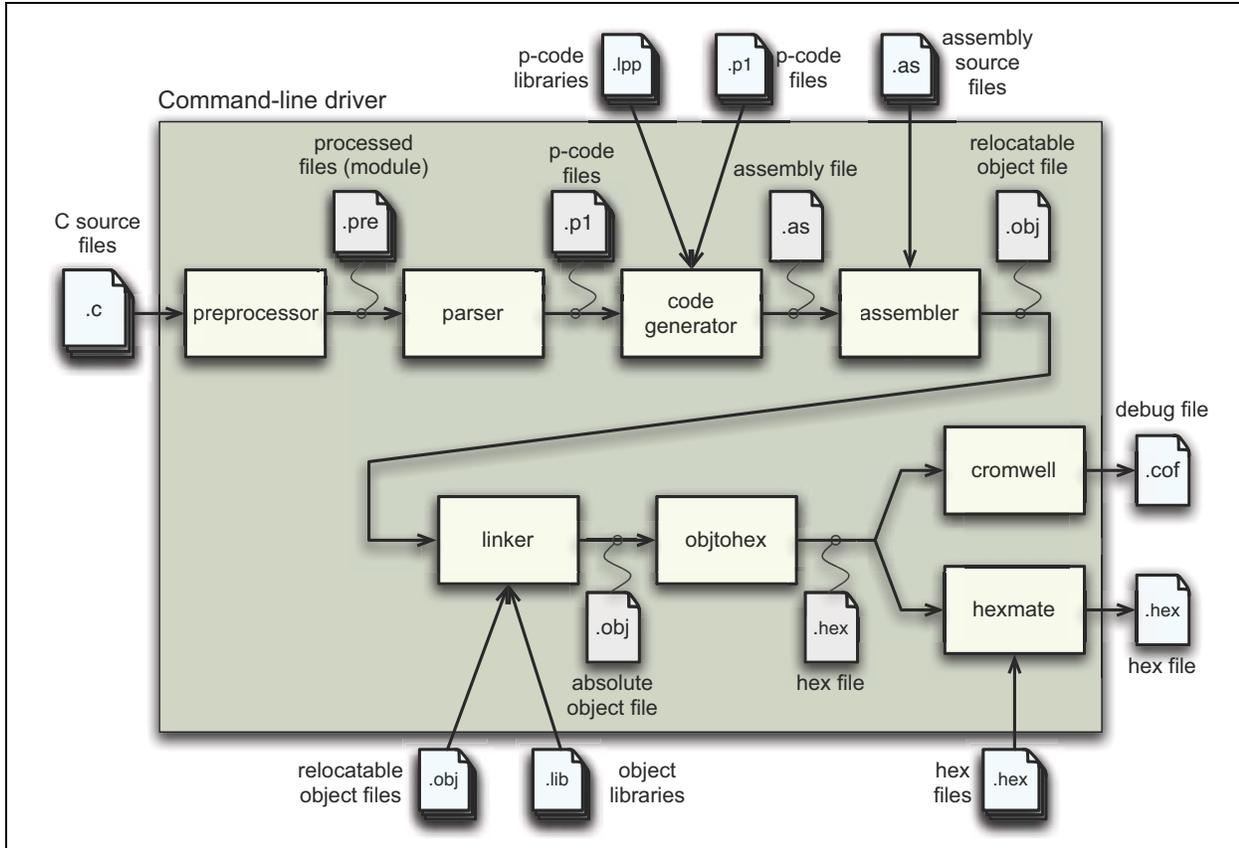
You can consider the large underlying box to represent the whole compiler, which is controlled by the command line driver, PICC. You may be satisfied just knowing that C source files (shown on the far left) are passed to the compiler, and the resulting output files (shown here as a HEX and COFF debug file on the far right) are produced, however internally there are many applications and temporary files being produced. An understanding of the internal operation of the compiler, while not necessary, does assist with using the tool.

The driver will call the required compiler applications. These applications are shown as the smaller boxed inside the large driver box. The temporary file produce by each application can also be seen in this diagram.

Table 2-2 lists the compiler applications. The names shown are the names of the exe-cutables, which can be found in the `bin` directory under the compiler's installation directory.

TABLE 2-2:   COMPILER APPLICATION NAMES

Name	Description
PICC	Command line driver; the interface to the compiler
CLIST	Text file formatter
CPP	The C preprocessor
P1	C code parser
CGPIC	Code generator
ASPIC	Assembler
HLINK	Linker
OBJTOHEX	Conversion utility to create HEX files
CROMWELL	Debug file converter
OBJTOHEX	Conversion utility to create HEX files
HEXMATE	HEX file utility
LIBR	Librarian
DUMP	Object file viewer
CREF	Cross reference utility

For example, C source files (`.c` files) are first passed to the C preprocessor, CPP. The output of this application are `.pre` files. These files are then passed to the parser appli-cation, P1, which produces a p-code file output with extension `.p1`. The applications are executed in the order specified and temporary files are used to pass the output of one application to the next.

The compiler can accept more than just C source files. Table 2-1 lists all the possible input file types, and these files can be seen in this diagram, on the top and bottom, being passed to different compilation applications. They are processed by these applications and then the application output joins the normal flow indicated in the diagram.

For example, assembly source files are passed straight to the assembler application[1] and are not processed at all by the code generator. The output of the assembler (an object file with `.obj` extension) is passed to the linker in the usual way. You can see that any p-code files (`.p1` extension) or p-code libraries (`.lpp` extension) that are supplied on the command line are initially passed to the code generator.

Other examples of input files include object files (`.obj` extension) and object libraries (`.lib` extension), both of which are passed initially to the linker, and even HEX files (`.hex` extension), which are passed to one of the utility applications, called HEXMATE, which is run right at the end of the compilation sequence.

Some of the temporary files shown in this diagram are actually preserved and can be inspected after compilation has concluded. There are also driver options to request that the compilation sequence stop after a particular application and the output of that application becomes the final output.

---

1. Assembly file will be preprocessed before being passed to the assembler if the -P option is selected.

FIGURE 2-2: MULTI-FILE COMPILATION
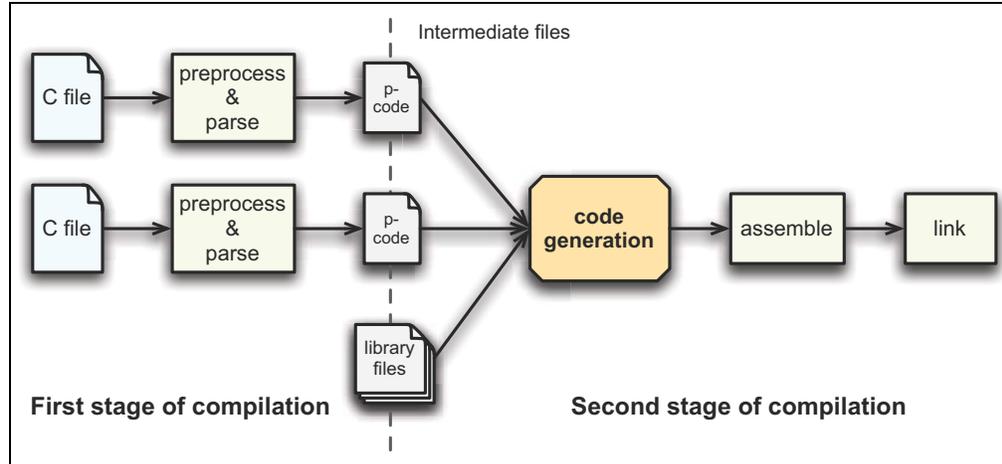
2.3.1 Single-step Compilation

Figure 2-1 showed us the files that are generated by each application and the order in which these applications are executed. However this does not indicate how these applications are executed when there is more than one source file being compiled.

Consider the case when there are two C source files that form a complete project and that are to be compiled, as is the case shown in Figure 2-2. If these files are called `main.c` and `io.c`, these could be compiled with a single command, such as:

```
PICC --chip=16F877A main.c io.c
```

This command will compile the two source files all the way to the final output, but internally we can consider this compilation as consisting of two stages.

The first stage involves processing of each source file separately, and generating some sort of intermediate file for each source file. The second stage involves combining all these intermediate files and further processing to form the final output. An intermediate file is a particular temporary file that is produced and marks the mid point between the first and second stage of compilation.

The intermediate file used by PICC is the p-code (`.p1` extension) file output by the parser, so there will be one p-code file produced for each C source file. As indicated in the diagram, CPP and then P1 are executed to form this intermediate file. (For clarity the CPP and P1 applications have been represented by the same block in the diagram.)

In the second stage, the code generator reads in all the intermediate p-code files an produces a single assembly file output, which is then passed to the subsequent applications that produce the final output.

The desirable attribute of this method of compilation is that the code generator, which is the main application that transforms from the C to the assembly domain, sees the entire project source code via the intermediate files.

Traditional compilers have always use intermediate files that are object files output by the assembler. These intermediate object files are then combined by the linker and further processed to form the final output. This method of compilation is shown in Figure 2-3 and shows that the code generator is executed once for each source file. Thus the code generator can only analyze that part of the project that is contained in the source file currently being compiled.

FIGURE 2-3: **THE TRADITIONAL COMPILATION SEQUENCE**
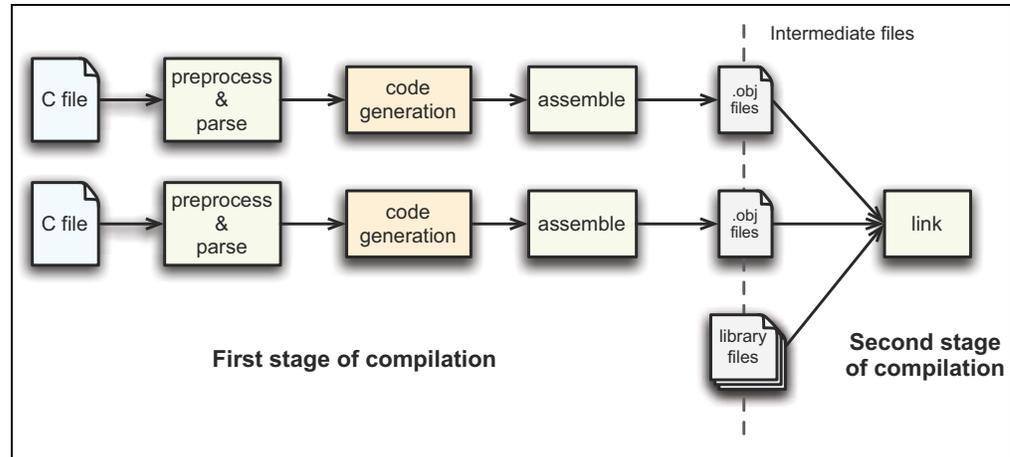
When compiling files of mixed types, this can still be achieved with just one invocation of the compiler driver. As discussed in **Section 2.3 "The Compilation Sequence"**, the driver will pass each input file to the appropriate compiler application.

For example, the files, `main.c`, `io.c`, `mdef.as` and `c_sb.lpp` are to be compiled. To perform this in a single step, the following command line could be used.

```
PICC --chip=16F877A main.c io.c mdef.as c_sb.lpp
```

As shown in Figure 2-1 and Figure 2-2, the two C files (`main.c` and `io.c`) will be compiled to intermediate p-code files; these, along with the p-code library file (`c_sb.lpp`) will be passed to the code generator. The output of the code generator, as well as the assembly source file (`mdef.as`), will be passed to the assembler.

The driver will re-compile all source files regardless of whether they have changed since the last build. IDEs, such as MPLAB IDE or HI-TIDE, and make utilities must be employed to achieve incremental builds, if desired. See also **Section 2.3.2 "Generating Intermediate Files"**.

Unless otherwise specified, a HEX file and Microchip COFF file are produced as the final output. All intermediate files remain after compilation has completed, but most other temporary files are deleted, unless you use the `--NODEL` option (see **Section 2.7.39 "--NODEL: Do not remove temporary files"**) which preserves all generated files except the run-time start-up file. Note that some generated files may be in a different directory to your project source files. See **Section 2.7.43 "--OUTDIR: Specify a directory for output files"** and **Section 2.7.41 "--OBJDIR: Specify a directory for intermediate files"** which can both control the destination for some output files.

2.3.2 Generating Intermediate Files

Make utilities and IDEs, such as MPLAB IDE and HI-TIDE, allow for an incremental build of projects that contain multiple source files. When building a project, they take note of which source files have changed since the last build and use this information to speed up compilation.

For example, if compiling two source files, but only one has changed since the last build, the intermediate file corresponding to the unchanged source file need not be regenerated.

The Universal Toolsuite plugin that integrates the compiler into MPLAB IDE is aware of the different compilation sequence employed by PICC and takes care of this for you. From MPLAB IDE you can select an incremental build (*Project->Build*), or fully re-build a project (*Project->Rebuild*).

If the compiler is being invoked using a make utility, it will need to be configured to recognized the different intermediate file format and the options used to generate the intermediate files. Make utilities typically call the compiler multiple times: once for each source file to generate an intermediate file, and once to perform the second stage compilation.

You may also wish to generate intermediate files to construct your own library files, although PICC is capable of constructing libraries in a single step, so this is typically not necessary. See **Section 2.7.44 "--OUTPUT= type: Specify Output File Type"** for more information on library creation.

The option `--PASS1` (**Section 2.7.45 "--PASS1: Compile to P-code"**) is used to tell the compiler that compilation should stop after the parser has executed. This will leave the p-code intermediate file behind on successful completion.

For example, the files `main.c` and `io.c` are to be compiled using a make utility. The command lines that the make utility should use to compile these files might be something like:

```
PICC --chip=16F877A --pass1 main.c
PICC --chip=16F877A --pass1 io.c
PICC --chip=16F877A main.p1 io.p1
```

If is important to note that the code generator needs to compile all p-code or p-code library files associated with the project in the one step. When using the `--PASS1` option the code generator is not being invoked, so the above command lines do not violate this requirement.

2.3.2.1   INTERMEDIATE FILES AND ASSEMBLY SOURCE

The intermediate file format associated with assembly source files is the same as that used in traditional compilers, i.e. an object file (`.obj` extension). Assembly files are never passed to the code generator and so the code generator technology does not alter the way these files are compiled.

The `-C` option (see **Section 2.7.1 "-C: Compile to Object File"**) is used to generate object files and halt compilation after the assembly step.

2.3.3   Compilation of Assembly Source

Since the code generator performs many tasks that were traditionally performed by the linker, there can be complications when assembly source is present in a project. Assembly files are traditionally processed after C code, but it is necessary to have this performed first so that specific information contained in the assembly code can be conveyed to the code generator.

The specific information passed to the code generator is discussed in more detail in **Section 3.9.4 "Interaction between Assembly and C Code"**.

When assembly source is present, the order of compilation is as shown in Figure 2-4.

FIGURE 2-4: **COMPILATION SEQUENCE WITH ASSEMBLY FILES**
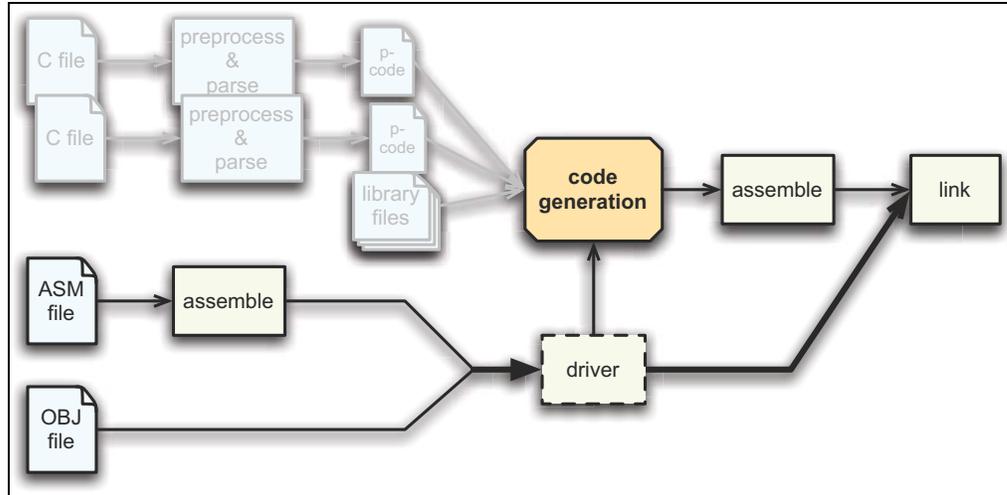
Any assembly source files are first assembled to form object files. These files, along with any other objects files that are part of the project, are then scanned by the command-line driver and information is then passed to the code generator when it subsequently builds the C files, as has been described earlier.

2.4 RUNTIME FILES

In addition to the C and assembly source files specified on the command line, there are also compiler-generated source files and pre-compiled library files which might be compiled into the project by the driver. These files contain:

- C Standard library routines
- Implicitly called arithmetic routines
- User-defined library routines
- The runtime startup code
- The powerup routine
- The `printf` routine.

Strictly speaking the power-up routine is neither compiler-generated source, nor a library routine. It is fully defined by the user, however as it is very closely associated with the runtime startup module, it is discussed with the other runtime files in the following sections.

2.4.1 Library Files

The names of the C standard library files appropriate for the selected target device, and other driver options, are determined by the driver and passed to the code generator and linker. P-code libraries (`.lpp` libraries) are used by the code generator, and object code libraries (`.lib` files) are used by the linker. Most library routines are derived from p-code libraries.

By default, PICC will search the `lib` directory under the compiler installation directory for library files that are required during compilation.

2.4.1.1 USING LIBRARY ROUTINES

Library functions or routines (and any associated variables) will be automatically linked into a program once they have been referenced in your source code. The use of a function from one library file will not include any other functions from that library. Only used library functions will be linked into the program output and consume memory.

Your program will require declarations for any functions or symbols used from libraries. These are contained in the standard C header (`.h`) files. Header files are not library files and the two files types should not be confused. Library files contain precompiled code, typically functions and variable definitions; the header files provide declarations (as opposed to definitions) for functions, variables and types in the library files, as well as other preprocessor macros.

```c
#include <math.h>     // declare function prototype for sqrt

void main(void)
{
  double i;

  // sqrt referenced; sqrt will be linked in from library file
  i = sqrt(23.5);
}
```

2.4.1.2   STANDARD LIBRARIES

The C standard libraries contain a standardised collection of functions, such as string, math and input/output routines. The range of these functions are described in **Chapter 7. "Library Functions"**.

These libraries also contain C routines that are implicitly called by the output code of the code generator. These are routines that perform tasks such as floating point operations, integer division and type conversions, and that may not directly correspond to a C function call in the source code.

The general form of the standard library names is `htpic`-*dc*.*ext*. The meaning of each field is described by:

- The processor type is always `pic`.
- The double type, *d*, is "-"for 24-bit doubles, and "d" for 32-bit doubles.
- Library Type is always "c".
- The extension is `.lpp` for p-code libraries, or `.lib` for relocatable object libraries.

2.4.1.3   USER-DEFINED LIBRARIES

User-defined libraries may be created and linked in with programs as required. Library files are more easy to manage and may result in faster compilation times, but must be compatible with the target device and options for a particular project. Several versions of a library may need to be created to allow it to be used for different projects.

Libraries can be created manually using the compiler and the librarian, `LIBR`. See **Section 6.2 "Librarian"** for more information on the librarian and creating library files using this application. Alternatively, library files can be created directly from the compiler by specifying a library output using the `--OUTPUT` option, see **Section 2.7.44 "--OUTPUT= type: Specify Output File Type"**.

User-created libraries that should be searched when building a project can be listed on the command line along with the source files.

As with Standard C library functions, any functions contained in user-defined libraries should have a declaration added to a header file. It is common practise to create one or more header files that are packaged with the library file. These header files can then be included into source code when required.

Library files specified on the command line are scanned first for unresolved symbols, so these files may redefined anything that is defined in the C standard libraries. See also **Section 3.11.1 "Replacing Library Modules"**.

2.4.2 Runtime Startup Code

A C program requires certain objects to be initialized and the processor to be in a particular state before it can begin execution of its function `main()`. It is the job of the *runtime startup* code to perform these tasks, specifically:

• Initialization of global variables assigned a value when defined
• Clearing of non-initialized global variables
• General setup of registers or processor state

Rather than the traditional method of linking in a generic, precompiled routine, HI-TECH C Compiler for PIC10/12/16 MCUs uses a more efficient method which actually determines what runtime startup code is required from the user's program.

Both the driver and code generator are involved in generating the runtime startup code. The driver takes care of device setup and this code is placed into a separate assembly startup module. The code generator handles initialization of the C environment, such as clearing uninitialized C variables and copying initialized C variables. This code is output along with the rest of the C program.

The runtime startup code is generated automatically every time you build a project. The file created by the driver may be deleted after compilation and this operation can be controlled with the `keep` suboption to the `--RUNTIME` option. The default operation of the driver is to keep the start up module, however if using MPLAB IDE to build, it requests that the file be deleted unless you indicate otherwise.

If the startup module is kept, it will be called `startup.as` and will be located in the current working directory. If you are using an IDE to perform the compilation the destination directory may be dictated by the IDE itself.

Generation of the runtime startup code is an automatic process which does not require any user interaction, however some aspects of the runtime code can be controlled, if required, using the `--RUNTIME` option. **Section 2.7.50 "--RUNTIME: Specify Runtime Environment"** describes the use of this option, and the following sections describes the functional aspects of the code contained in this module and its effect on program operation.

The runtime startup code is executed before `main()`, but If you require any special initialization to be performed immediately after reset, you should use power-up feature described later in **Section 2.4.3 "The Powerup Routine"**.

2.4.2.1 INITIALIZATION OF OBJECTS

One task of the runtime startup code is to ensure that any initialized variables contain their initial value before the program begins execution. Initialized variables are those which are not `auto` objects and which are assigned an initial value in their definition, for example `input` in the following example.

```
int input = 88;
void main(void) { ...
```

Such initialized objects have two components: their initial value (0x0088 in the above example) stored in program memory (i.e. placed in the HEX file), and space for the variable reserved in RAM it will reside and be accessed during program execution (runtime).

The psects used for storing these components are described in **Section 3.7.1 "Compiler-generated Psects"**.

The runtime startup code will copy all the blocks of initial values from program memory to RAM so the variables will be contain the correct values before `main()` is executed.

Since `auto` objects are dynamically created, they require code to be positioned in the function in which they are defined to perform their initialization. It is possible that the initial value of an `auto` object may change on each instance of the function and so the initial values cannot be stored in program memory and copied. As a result, initialized `auto` objects are not considered by the runtime startup code but are instead initialized by assembly code in each function output.

> **Note:** Initialized `auto` variables can impact on code performance, particularly if the objects are large in size. Consider using global or `static` objects instead.

Variables whose contents should be preserved over a reset, or even power off, should be qualified with the `persistent` qualifier, see **Section 3.3.11.1 "Persistent Type Qualifier"**. Such variables are linked at a different area of memory and are not altered by the runtime startup code in any way.

2.4.2.2 CLEARING OBJECTS

Those non-`auto` objects which are not initialized must be cleared before execution of the program begins. This task is also performed by the runtime startup code.

Uninitialized variables are those which are not `auto` objects and which are not assigned a value in their definition, for example `output` in the following example.

```
int output;
void main(void) { ...
```

Such uninitialized objects will only require space to be reserved in RAM where they will reside and be accessed during program execution (runtime).

The psects used for storing these components are described in **Section 3.7.1 "Compiler-generated Psects"** and typically have a name based on the initialism "bss" (Block Started by Symbol).

The runtime startup code will clear all the memory location occupied by uninitialized variables so they will contain zero before `main()` is executed.

Variables whose contents should be preserved over a reset should be qualified with `persistent`. See **Section 3.3.11.1 "Persistent Type Qualifier"** for more information. Such variables are linked at a different area of memory and are not altered by the runtime startup code in any way.

2.4.2.3 STATUS REGISTER PRESERVATION

The `resetbits` suboption of the `--RUNTIME` option (see **2.7.50 "--RUNTIME: Specify Runtime Environment"**) preserves some of the bits in the `STATUS` register before being clobbered by the remainder of the runtime startup code. The state of these bits can be examined after recovering from a reset condition to determine the cause of the reset.

The entire `STATUS` register is saved to an assembly variable `___resetbits`. This variable can be accessed from C code using the declaration:

```
extern unsigned char __resetbits;
```

The assembly equates `___powerdown` and `___timeout` represent the bit address of the powerdown and timeout bits within the `STATUS` register and can be used if required. These can be accessed from C code using the declarations:

```
extern bit __powerdown;
extern bit __timeout;
```

See **Section 2.8 "MPLAB IDE Universal Toolsuite Equivalents"** for use of this option in MPLAB IDE.

2.4.3   The Powerup Routine

Some hardware configurations require special initialization, often within the first few instruction cycles after reset. To achieve this there is a hook to the reset vector provided via the *powerup* routine.

This routine can be supplied in a user-defined assembler module that will be executed immediately after reset. An template powerup routine is provided in the file `pow-erup.as` which is located in the `sources` directory of your compiler distribution. Refer to comments in this file for more details.

The file should be copied to your working directory, modified and included into your project as a source file. No special linker options or other code is required. The compiler will detect if you have defined a powerup routine and will automatically use it, provided the code in this routine is contained in a psect called `powerup`.

For correct operation (when using the default compiler-generated runtime startup code), the code must end with a `GOTO` instruction to the label called `start`. As with all user-defined assembly code, it must take into consideration program memory paging and/or data memory banking, as well as any applicable errata issues for the device you are using. The program's entry point is already defined by the runtime startup code, so this should not be specified in the power-up routine with the `END` directive (if used). See **Section 4.3.9.2 "END"** for more information on this assembler directive.

2.4.4   The `printf` Routine

The code associated with the `printf` function is not found in the library files. The `printf()` function is generated from a special C template file that is customized after analysis of the user's C code. See **Section "PRINTF, VPRINTF"** for more information on the `printf` library function.

The template file is found in the `lib` directory of the compiler distribution and is called `doprnt.c`. It contains a minimal implementation of the `printf()` function, but with the more advanced features included as conditional code which can be utilized via preprocessor macros that are defined when it is compiled.

The parser and code generator analyze the C source code, searching for calls to the `printf` function. For all calls, the placeholders that were specified in the `printf()` format strings are collated to produce a list of the desired functionality of the final function. The `doprnt.c` file is then preprocessed with the those macros specified by the preliminary analysis, thus creating a custom `printf()` function for the project being compiled. After parsing, the p-code output derived from `doprnt.c` is then combined with the remainder of the C program in the final code generation step.

For example, if a program contains one call to `printf()`, which looks like:

```
printf("input is: %d");
```

The compiler will note that only the `%d` placeholder is used and the `doprnt.c` module that is linked into the program will only contain code that handles printing of decimal integers.

Consider now that the code is changed and another call to `printf()` is added. The new call looks like:

```
printf("output is %6d");
```

Now the compiler will detect that additional code to handle printing decimal integers to a specific width must be enabled as well.

As more features of `printf()` are detected, the size of the code generated for the `printf()` function will increase.

If the format string in a call to `printf()` is not a string literal as above, but is rather a pointer to a string, then the compiler will not be able to reliably predict the `printf()` usage, and so it forces a more complete version of `printf()` to be generated.

However, even without being able to scan `printf()` placeholders, the compiler can still make certain assumptions regarding the usage of the function. In particular, the compiler can look at the number and type of the additional arguments to `printf ()` (those following the format string expression) to determine which placeholders could be valid. This enables the size and complexity of the generated `printf()` routine to be kept to a minimum even in this case.

For example, if `printf()` was called as follows:

```
printf(myFormatString, 4, 6);
```

the compiler could determine that, for example, no floating point placeholders are required and omit these from being included in the `printf()` function output. As the arguments after the format string are non-prototyped parameters, their type must match that of the placeholders.

No aspect of this operation is user-controllable (other than by adjusting the calls to `printf()`), however the actual `printf()` code used by a program can be observed. If compiling a program using `printf()`, the driver will leave behind the pre-processed version of `doprnt.c`. This module, called `doprnt.pre` in your working directory, will show the C code that will actually be contained in the `printf` routine. As this code has been pre-processed, indentation and comments will have been stripped out as part of the normal actions taken by the C pre-processor.

2.5   DEBUGGING INFORMATION

Several driver options and output files assist with code development and allow source-level debugging of the output code. These are described in the following sections.

2.5.1   Output File Formats

The compiler is able to directly produce a number of the output file formats which are used by PIC10/12/16 development tools.

The default behavior of PICC is to produce a, Microchip format COFF and Intel HEX output. Unless changed by a driver option, the base names of these files will be the project name. See **Section 2.2.1 "Output Files"** for more information on how this base name is derived.

The COFF file is used by debuggers to obtain debugging information about the project.

Table 2-13 shows all output format options available with PICC using the `--OUTPUT` option. The File Type column lists the filename extension which will be used for the output file.

2.5.1.1   SYMBOL FILES

PICC creates two symbol files which are used to generate the debug output files, such as COFF and ELF files. These are the SYM file (`.sym` extension), produced by the linker, and the SDB file (`.sdb` extension) produced by the code generator.

The SDB file contains type information, and the SYM file contains address information.These two files, in addition to the HEX file, are combined by the CROMWELL application (see **Section 6.5 "Cromwell"**) to produce the output debug files, such as the COFF file.

2.5.2 Support Files

Two valuable files produced by the compiler are the assembly list file, produced by the assembler, and the map file, produced by the linker.

The compiler options `--ASMLIST` (**Section 2.7.17 "--ASMLIST: Generate Assembler List Files"**) generates a list files, and the `-M` option (**Section 2.7.8 "-M: Generate Map File"**) specifies generation of a map file.

The assembly list file contains the mapping between the original source code and the generated assembly code. It is useful for information such as how C source was encoded, or how assembly source may have been optimized. It is essential when confirming if compiler-produced code that accesses objects is atomic, and shows the psects in which all objects and code are placed. See **Section 4.4 "Assembly List Files"** for more information on the contents of this file.

The map file shows information relating to where objects were positioned in memory. It is useful for confirming if user-defined linker options were correctly processed, and for determining the exact placement of objects and functions. It also shows all the unused memory areas in a devices and memory fragmentation. See **Section 5.4 "Map Files"** for complete information on the contents of this file.

2.6 COMPILER MESSAGES

All compiler applications, including the command-line driver, PICC, use textual messages to report feedback during the compilation process. A centralized messaging system is used to produce the messages which allows consistency during all stages of the compilation process.

2.6.1 Messaging Overview

A message is referenced by a unique number which is passed to the messaging system by the compiler application that needs to convey the information. The message string corresponding to this number is obtained from Message Description Files (MDF) which are stored in the `dat` directory in the compiler's installation directory.

When a message is requested by a compiler application, its number is looked up in the MDF that corresponds to the currently selected language. The language of messages can be altered as discussed in **Section 2.6.2 "Message Language"**.

Once found, the alert system can read the message type and the string to be displayed from the MDF. There are several different message types which are described in **Section 2.6.3 "Message Type"** and the type can be overridden by the user, as described in the same section.

The user is also able to set a threshold for warning message importance, so that only those which the user considers significant will be displayed. In addition, messages with a particular number can be disabled. A pragma can also be used to disabled a particular message number within specific lines of code. These methods are explained in **Section 2.6.5.1 "Disabling Messages"**.

Provided the message is enabled and it is not a warning messages whose level is below the current warning threshold, the message string will be displayed.

In addition to the actual message string, there are several other pieces of information that may be displayed, such as the message number, the name of the file for which the message is applicable, the file's line number and the application that requested the message, etc.

If a message is an error, a counter is incremented. After a certain number of errors has been reached, compilation of the current module will cease. The default number of errors that will cause this termination can be adjusted by using the `--ERRORS` option,

see **Section 2.7.28 "--ERRORS: Maximum Number of Errors"**. This counter is reset for each compiler application, thus specifying a maximum of five errors will allow up to five errors from the parser, five from the code generator, five from the linker, five from the driver, etc.

Although the information in the MDF can be modified with any text editor, this is not recommended. Message behavior should only be altered using the options and pragmas described in the following sections.

2.6.2 Message Language

PICC Supports more than one language for displayed messages. There is one MDF for each language supported.

Under Windows, the default language can be specified when installing the compiler.

The default language may be changed on the command line using the `--LANG` option, see **Section 2.7.34 "--LANG: Specify the Language for Messages"**. Alternatively it may be changed permanently by using the `--LANG` option together with the `--SETUP` option which will store the default language in either the registry, under Windows, or in a configuration file on other systems. On subsequent builds the default language used will be that specified.

Table shows the MDF applicable for the currently supported languages.

TABLE 2-3:   SUPPORTED LANGUAGES

Language	MDF name
English	`en_msgs.txt`
German	`de_msgs.txt`
French	`fr_msgs.txt`

If a language other than English is selected, and the message cannot be found in the appropriate non-English MDF, the alert system tries to find the message in the English MDF. If an English message string is not present, a message similar to:

```
error/warning (*) generated, but no description available
```

where * indicates the message number that was generated that will be printed; otherwise, the message in the requested language will be displayed.

2.6.3 Message Type

There are four types of message. These are described below along with the compiler's behavior when encountering a message of each type.

Advisory Messages convey information regarding a situation the compiler has encountered or some action the compiler is about to take. The information is being displayed "for your interest" and typically require no action to be taken. Compilation will continue as normal after such a message is issued.

Warning Messages indicate source code or some other situation that can be compiled, but is unusual and may lead to runtime failure of the code. The code or situation that triggered the warning should be investigated; however, compilation of the current module will continue, as will compilation of any remaining modules.

Error Messages indicate source code that is illegal or that compilation of this code cannot take place. Compilation will be attempted for the remaining source code in the current module, but no additional modules will be compiled and the compilation process will then conclude.

Fatal Error Messages indicate a situation that cannot allow compilation to proceed and which required the compilation process to stop immediately.

2.6.4   Message Format

By default, messages are printed in a human-readable format. This format can vary from one compiler application to another, since each application reports information about different file formats.

Some applications, for example the parser, are typically able to pinpoint the area of interest down to a position on a particular line of C source code, whereas other applications, such as the linker, can at best only indicate a module name and record number, which is less directly associated with any particular line of code. Some messages relate to issues in driver options which are in no way associated with any source code.

There are several ways of changing the format in which message are displayed, which are discussed below.

The driver option -E (with or without a filename) alters the format of all displayed messages. See **Section 2.7.3 "-E: Redirect Compiler Errors to a File"**. Using this option produces messages that are better suited to machine parsing, and are less user-friendly. Typically each message is displayed on a single line. The general form of messages produced when using the -E option is:

```
filename  line: (message number)  message string  (type)
```

The -E option also has another effect. When used the driver first checks to see if special environment variables have been set. If so, the format dictated by these variables are used as a template for all messages produced by all compiler applications. The names of these environment variables are given in Table 2-4.

TABLE 2-4:   MESSAGING ENVIRONMENT VARIABLES

Variable	Effect
HTC_MSG_FORMAT	All advisory messages
HTC_WARN_FORMAT	All warning messages
HTC_ERR_FORMAT	All error and fatal error messages

The value of these environment variables are strings that are used as templates for the message format. Printf-like placeholders can be placed within the string to allow the message format to be customized. The placeholders and what they represent are indicated in Table 2-5.

TABLE 2-5:   MESSAGING PLACEHOLDERS

Placeholder	Replacement
%a	application name
%c	column number
%f	filename
%l	line number
%n	message number
%s	message string (from MDF)

If these options are used in a DOS batch file, two percent characters will need to be used to specify the placeholders, as DOS interprets a single percent character as an argument and will not pass this on to the compiler. For example:

```
SET HTC_ERR_FORMAT="file %%f: line %%l"
```

Environment variables, in turn, may be overridden by the driver options: `--MSGFOR-MAT`, `--WARNFORMAT` and `--ERRFORMAT`, see **Section 2.7.27 "--ERRFORMAT: Define Format for Compiler Messages"**. These options take a string as their argument. The option strings are formatted, and can use the same placeholders, as their variable counterparts.

For example, a project is compiled, but, as shown, produces a warning from the parser and an error from the linker (numbered 362 and 492 respectively).

```
main.c: main()
    17: ip = &b;
              ^ (362) redundant "&" applied to array (warning)
(492) attempt to position absolute psect "text" is illegal
```

Notice that the parser message format identifies the particular line and position of the offending source code.

If the `-E` option is now used and the compiler issues the same messages, the compiler will output.

```
main.c: 12: (362) redundant "&" applied to array (warning)
(492) attempt to position absolute psect "text" is illegal (error)
```

The user now uses the `--WARNFORMAT` in the following fashion.

```
--WARNFORMAT="%a %n %l %f %s"
```

When recompiled, the following output will be displayed.

```
parser 362 12 main.c redundant "&" applied to array
(492) attempt to position absolute psect "text" is illegal (error)
```

Notice that the format of the warning was changed, but that of the error message was not. The warning format now follows the specification of the environment variable. The application name (parser) was substituted for the `%a` placeholder, the message number (362) substituted the `%n` placeholder, etc.

2.6.5 Changing Message Behavior

Both the attributes of individual messages and general settings for the messaging system can be modified during compilation. There are both driver options and C pragmas that can be used to achieve this.

2.6.5.1 DISABLING MESSAGES

Each warning message has a default number indicating a level of importance. This number is specified in the MDF and ranges from -9 to 9. The higher the number, the more important the warning.

Warning messages can be disabled by adjusting the warning level threshold using the `--WARN` driver option, see **Section 2.7.59 "--WARN: Set Warning Level"**. Any warnings whose level is below that of the current threshold are not displayed.

The default threshold is 0 which implies that only warnings with a warning level of 0 or higher will be displayed by default. The information in this option is propagated to all compiler applications, so its effect will be observed during all stages of the compilation process.

Warnings may also be disabled by using the `--MSGDISABLE` option, see **Section 2.7.37 "--MSGDISABLE: Disable Warning Messages"**. This option takes a *comma*-separated list of message numbers. Those warnings listed are disabled and will never be issued, regardless of the current warning level threshold. This option cannot be used to disable error messages.

Some warning messages can also be disabled by using the `warning` pragma. This pragma will only affect warnings that are produced by either the parser or the code generator, i.e. errors directly associated with C code. See **Section 3.10.3.9 "The #pragma warning Directive"** for more information on this pragma.

Error messages can also be disabled, however a more verbose form of the command is required to confirm the action. To specify an error message number in the `--MSG-DISABLE` command, the number must be followed by `:off` to ensure that it is disabled. For example: `--MSGDISABLE=195:off` will disable error number 195.

> **Note:** Disabling error or warning messages in no way fixes the condition which triggered the message. Always use extreme caution when exercising these options.

2.6.5.2   CHANGING MESSAGE TYPES

It is also possible to change the type of some messages. This can only be done for messages generated by the parser or code generator. See **Section 3.10.3.9 "The #pragma warning Directive"** for more information on this pragma.

2.7   PICC DRIVER OPTION DESCRIPTIONS

Most aspects of the compilation can be controlled using the command-line driver, PICC. The driver will configure and execute all required applications, such as the code generator, assembler and linker.

PICC recognizes the compiler options which are tabled below and explained in detail in the following sections. The case of the options is not important, however command shells in most operating systems are case sensitive when it comes to names of files.

TABLE 2-6:   DRIVER OPTIONS

Option	Meaning
-C	Compile to object file and stop
-D*macro*	Define preprocessor macro symbol
-E*filename*	Redirect compile errors
-G[*filename*]	Generate symbolic debug information
-I*path*	Specify include path
-L*argument*	Set linker option
-M[*filename*]	Generate map file
-N*number*	Specify identifier length
-O*file*	Specify output filename and type
-P	Preprocess assembly source
-Q	Quiet mode
-S	Compile to assembly file and stop
-U*macro*	Undefine preprocessor macro symbol
-V	Verbose mode
-X	Strip local symbols
--ADDRQUAL=*qualifier*	Specify address space qualifier handling
--ASMLIST	Generate assembly list file
--CHAR=*type*	Default character type (defunct)
--CHECKSUM=*specification*	Calculate a checksum and store the result in program memory
--CHIP=*device*	Select target device
--CHIPINFO	Print device information

TABLE 2-6: DRIVER OPTIONS (CONTINUED)

Option	Meaning
--CODEOFFSET=*value*	Specify ROM offset address
--CR=*file*	Generate cross reference file
--DEBUGGER=*type*	Set debugger environment
--DOUBLE=*size*	Size of double type
--ECHO	Echo command line
--ERRFORMAT=*format*	Set error format
--ERRORS=*number*	Set maximum number of errors
--FILL=*specification*	Specify a ROM-fill value for unused memory
--FLOAT=*size*	Size of float type
--GETOPTION=*argument*	Get advanced options
--HELP=*option*	Help
--IDE=*name*	Set development environment
--LANG=*language*	Specify language
--MEMMAP=*type*	Display memory map
--MODE=*mode*	Choose operating mode
--MSGDISABLE=*list*	Disable warning messages
--MSGFORMAT=*specification*	Set advisory message format
--NODEL	Do not remove temporary files
--NOEXEC	Do not execute compiler applications
--OBJDIR=*path*	Set object files directory
--OPT=*optimizations*	Control optimization
--OUTDIR=*path*	Set output directory
--OUTPUT=*path*	Set output formats
--PASS1	Produce HI-TECH intermediate p-code file and stop
--PRE	Produce preprocessed source files and stop
--PROTO	Generate function prototypes
--RAM=*ranges*	Adjust RAM ranges
--ROM=*ranges*	Adjust ROM ranges
--RUNTIME=*options*	Specify runtime options
--SCANDEP	Scan for dependencies
--SERIAL=*specification*	Insert a hexadecimal code or serial number
--SETOPTION=*argument*	Set advanced options
--SETUP=*specification*	Setup the compiler
--SHROUD	Shroud (obfuscate) generated p-code files
--STRICT	Use strict ANSI keywords
--SUMMARY=*type*	Summary options
--TIME	Report compilation times
--VER	Show version information
--WARN=*number*	Set warning threshold level
--WARNFORMAT=*specification*	Set warning format

2.7.0.1   OPTION FORMATS

All single letter options are identified by a leading *dash* character, "-", e.g. -C. Some single letter options specify an additional data field which follows the option name immediately and without any *whitespace*, e.g. -Ddebug. In this manual options are written in upper case and suboptions are in lower case.

Multi-letter, or word, options have two leading *dash* characters, e.g. --ASMLIST. (Because of the double *dash*, the driver can determine that the option --DOUBLE, for example, is not a -D option followed by the argument OUBLE.)

Some of these word options use suboptions which typically appear as a *comma* -separated list following an *equal* character, =, e.g. --OUTPUT=hex,cof. The exact format of the options varies and are described in detail in the following sections.

Some commonly used suboptions include default, which represent the default specification that would be used if this option was absent altogether; all, which indicates that all the available suboptions should be enabled as if they had each been listed; and none, which indicates that all suboptions should be disabled. For example:

--OPT=none

will turn off all optimizers.

Some suboptions may be prefixed with a plus character, +, to indicate that they are in addition to the other suboptions present, or a minus character "-", to indicate that they should be excluded. For example:

--OPT=default,-asm

indicates that the default optimization be used, but that the assembler optimizer should be disabled. If the first character after the equal sign is + or -, then the default keyword is implied. For example:

--OPT=-asm

is the same as the previous example.

See the --HELP option, **Section 2.7.32 "--HELP: Display Help"**, for more information about options and suboptions.

2.7.1   -C: Compile to Object File

The -C option is used to halt compilation after executing the assembler, leaving a relocatable object file as the output. It is frequently used when compiling assembly source files using a make utility.

See **Section 2.3.2 "Generating Intermediate Files"** for more information on generating and using intermediate files.

2.7.2   -D: Define Macro

The `-D` option is used to define a preprocessor macro on the command line, exactly as if it had been defined using a `#define` directive in the source code. This option may take one of two forms, `-Dmacro` which is equivalent to:

```
#define macro  1
```

placed at the top of each module compiled using this option, or `-Dmacro=` *text* which is equivalent to:

```
#define  macro  text
```

where *text* is the textual substitution required. Thus, the command:

```
PICC --CHIP=16F877AA -Ddebug -Dbuffers=10 test.c
```

will compile `test.c` with macros defined exactly as if the C source code had included the directives:

```
#define  debug    1
#define  buffers 10
```

Defining macros as C string literals requires bypassing any interpretation issues in the operating system that is being used. To pass the C string, `"hello world"`, (including the *quote* characters) in the Windows environment, use: `"-DMY_STRING=\\\"hello world\\\""` (you must include the *quote* characters around the entire option as there is a *space* character in the macro definition). Under Linux or Mac OS X, use: `-DMY_STRING=\"hello\ world\"`.

See **Section 2.8 "MPLAB IDE Universal Toolsuite Equivalents"** for use of this option in MPLAB IDE.

2.7.3   -E: Redirect Compiler Errors to a File

This option has two purposes. The first is to change the format of displayed messages. The second is to optionally allow messages to be directed to a file as some editors do not allow the standard command line redirection facilities to be used when invoking the compiler.

The general form of messages produced with the `-E` option in force is:

```
filename  line_number:  (message number)  message string  (type)
```

If a filename is specified immediately after `-E`, it is treated as the name of a file to which all messages (errors, warnings etc) will be printed. For example, to compile `x.c` and redirect all errors to `x.err`, use the command:

```
PICC --CHIP=16F877AA -Ex.err x.c
```

The `-E` option also allows errors to be appended to an existing file by specifying an addition character, `+`, at the start of the error filename, for example:

```
PICC --CHIP=16F877AA -E+x.err y.c
```

If you wish to compile several files and combine all of the errors generated into a single text file, use the `-E` option to create the file then use `-E+` when compiling all the other source files. For example, to compile a number of files with all errors combined into a file called `project.err`, you could use the `- E` option as follows:

```
PICC --CHIP=16F877AA -Eproject.err -O --PASS1 main.c
PICC --CHIP=16F877AA -E+project.err -O --PASS1 part1.c
PICC --CHIP=16F877AA -E+project.err -C asmcode.as
```

Section 2.6 "Compiler Messages" has more information regarding this option as well as an overview of the messaging system and other related driver options.

2.7.4   -G: Generate Source-level Symbol File

The `-G` option allows specification of the filename used for the *source-level symbol file* (.sym extension) for use with supported debuggers and simulators such as HI-TIDE™ and MPLAB IDE. See also **Section 2.5 "Debugging Information"**.

If no filename is given, the symbol file will have the same base name as the project name (see **Section 2.2 "Invoking the Compiler"**), and an extension of `.sym`. For example the option `-Gtest.sym` generates a symbol file called `test.sym`. Symbol files generated using the `-G` option include source-level information for use with source-level debuggers.

2.7.5   -I: Include Search Path

Use `-I` to specify an additional directory to search for header files which have been included using the `#include` directive. The directory can either be an absolute or relative path. The `-I` option can be used more than once if multiple directories are to be searched.

The compiler's `include` directory containing all standard header files is always searched, even if no `-I` option is present. If header filenames are specified using *quote* characters rather than *angle brackets*, as in `#include "lcd.h"`, then the current working directory is searched in addition to the compiler's `include` directory. Note that if compiling within MPLAB IDE, the search path is relative to the output directory, not the project directory.

These default search paths are searched after any user-specified directories have been searched. For example:

```
PICC --CHIP=16F877AA -C -Ic:\include -Id:\myapp\include test.c
```

will search the directories `c:\include` and `d:\myapp\include` for any header files included into the source code, then search the default include directory.

This option has no effect for files that are included into assembly source using the assembly `INCLUDE` directive. See **Section 4.3.10.3 "INCLUDE"**.

See **Section 2.8 "MPLAB IDE Universal Toolsuite Equivalents"** for use of this option in MPLAB IDE.

2.7.6   -L: Scan Library

The `-L` option is used to specify additional libraries which are to be scanned by the linker. Libraries specified using the `-L` option are scanned before the standard C library, allowing additional versions of standard library functions to be accessed.

The argument to -L is a library keyword to which the prefix `pic` ; numbers representing the processor range, number of ROM pages and the number of RAM banks; and the suffix `.lib` are added.

Thus the option `-Ll` when compiling for a 16F877A will, for example, scan the library `pic42c-l.lib` and the option `-Lxx` will scan a library called `pic42c-xx.lib`.

All libraries must be located in the `lib` directory of the compiler installation directory.

As indicated, the argument to the `-L` option is *not* a complete library filename. If you wish the linker to scan libraries whose names do not follow the above naming convention or whose locations are not in the `lib` subdirectory, simply include the libraries' names on the command line along with your source files, or add these to your project.

2.7.7 -L-: Adjust Linker Options Directly

The `-L` driver option can be used to specify an option which will be passed directly to the linker. If `-L` is followed immediately by text starting with a dash character "`-`", the text will be passed directly to the linker without being interpreted by PICC. If the `-L` option is not followed immediately by a dash character, it is assumed the option is the library scan option, **Section 2.7.6 "-L: Scan Library"**.

For example, if the option `-L-N` is specified, the `-N` option will be passed on to the linker without any subsequent interpretation by the driver. The linker will then process this option, when, and if, it is invoked, and perform the appropriate operation.

Take care with command-line options. The linker cannot interpret command-line driver options; similarly the driver cannot interpret linker options. In most situations, it is always the command-line driver, PICC, that is being executed. If you need to add alternate linker settings in the _Linker_ tab in the _Project>MPLAB Build options..._ dialogue, you must add *driver* options (not linker options). These driver options will be used by the driver to generate the appropriate linker options during the linking process. The `-L` option is a means of allowing a linker option to be specified via a driver option.

The `-L` option is especially useful when linking code which contains non-standard program sections (or psects), as may be the case if the program contains hand-written assembly code which contains user-defined psects (see **4.3.9.3 "PSECT"**), or C code which uses the `#pragma psect` directive (see **3.10.3.6 "The #pragma psect Directive"**). Without this `-L` option, it would be necessary to invoke the linker manually to allow the linker options to be adjusted.

This option can also be used to replace default linker options. If the string starting from the first character after the `-L` up to the first equal character, "=", matches a psect or class name in the default options, then (the reference to the psect or class name in the default option, and the remainder of that option, are deleted) that default linker option is replaced by the option specified by the `-L`. For example, if a default linker option was:

```
-preset_vec=00h,intentry,init,end_init
```

the driver option `-L-pinit=100h` would result in the following options being passed to the linker: `-pinit=100h -preset_vec=00h`. Note the `end_init` linker option has been removed entirely. If there are no characters following the first equal character in the `-L` option, then no replacement will be made for the default linker options that will be deleted. For example, the driver option `-L-pinit=` will adjust the default options passed to the linker, as above, but the `-pinit` linker option would be removed entirely.

No warning is generated if such a default linker option cannot be found. The default option that you are deleting or replacing must contain an equal character.

2.7.8 -M: Generate Map File

The `-M` option is used to request the generation of a map file. The map is generated by the linker and includes detailed information about where objects are located in memory. See **Section 5.4 "Map Files"** for information regarding the content of these files.

If no filename is specified with the option, then the name of the map file will have the project name (see **Section 2.2.1 "Output Files"**), with the extension `.map`.

This option is on by default when compiling from within MPLAB IDE and using the HI-TECH Universal Toolsuite.

2.7.9   -N: Identifier Length

This option allows the C identifier length to be increased from the default value of 31. Valid sizes for this option are from 32 to 255. The option has no effect for all other values.

This option also controls the length of identifiers used by the preprocessor, such as macro names. The default length is also 31, and can be adjusted to a maximum of 255.

See **Section 2.8 "MPLAB IDE Universal Toolsuite Equivalents"** for use of this option in MPLAB IDE.

2.7.10   -O: Specify Output File

This option allows the basename of the output file(s) to be specified. If no -O option is given, the base name of output file(s) will be the same as the project name, see **Section 2.2.1 "Output Files"**. The files whose names are affected by this option are those files that are not directly associated with any particular source file, such as the HEX file, MAP file and SYM file.

The -O option can also change the directory in which the output file is located by including the required path before the filename. This will then also specify the output directory for any files produced by the linker or subsequently run applications. Any relative paths specified are with respect to the current working directory.

For example, if the option -Oc:\project\output\first is used, the MAP and HEX file, etc, will use the base name first, and will be placed in the directory c:\project\output.

Any extension supplied with the filename will be ignored.

The options that specify MAP file creation (-M, see **Section 2.7.8 "-M: Generate Map File"**), and SYM file creation (-G, see **Section 2.7.4 "-G: Generate Source-level Symbol File"**) override any name or path information provided by -O relevant to the MAP and SYM file.

To change the directory in which all output and intermediate files are written, use the --OUTDIR option, see Section **Section 2.7.43 "--OUTDIR: Specify a directory for output files"**. Note that if -O specifies a path which is inconsistent with the path specified in the --OUTDIR option, this will result in an error.

2.7.11   -P: Preprocess Assembly Files

The -P option causes assembler source files to be preprocessed before they are assembled, thus allowing the use of preprocessor directives, such as #include, and C-style comments with assembler code.

By default, assembler files are not preprocessed.

See **Section 2.8 "MPLAB IDE Universal Toolsuite Equivalents"** for use of this option in MPLAB IDE.

2.7.12   -Q: Quiet Mode

This option places the compiler in a quiet mode which suppresses the Microchip Technology Incorporated copyright notice from being displayed.

2.7.13   -S: Compile to Assembler Code

The -S option stops compilation after generating an assembly output file. One assembly file will be generated for all the C source code, including p-code library code.

The command:

```
PICC --CHIP=16F877A -S test.c
```

will produce an assembly file called test.as which contains the assembly code generated from test.c. The generated file is valid assembly code could be passed to PICC as a source file, however this should only be done for exploratory reasons. To take advantage of the benefits of the compilation technology in the compiler, it must compile and link all the C source code in a single step. See the --PASS1 option (**Section 2.7.45 "--PASS1: Compile to P-code"**) to generate intermediate files if you wish to compile code using a two step process or use intermediate files.

This option is useful for checking assembly code output by the compiler. The file produced by this option differs to that produced by the --ASMLIST option (see **Section 2.7.17 "--ASMLIST: Generate Assembler List Files"**) in that it does not contain op-codes or addresses and it may be used as a source file in subsequent compilations. The assembly list file is more human readable, but is not a valid assembly source file.

2.7.14   -U: Undefine a Macro

The -U option, the inverse of the -D option, is used to undefine predefined macros. This option takes the form -U*macro*, where *macro* is the name of the macro to be undefined

The option, -Udraft, for example, is equivalent to:

```
#undef  draft
```

placed at the top of each module compiled using this option.

See **Section 2.8 "MPLAB IDE Universal Toolsuite Equivalents"** for use of this option in MPLAB IDE.

2.7.15   -V: Verbose Compile

The -V option specifies verbose compilation. When used the compiler will display the command lines used to invoke each of the compiler applications described in **Section 2.3 "The Compilation Sequence"**.

Displayed will be the name of the compiler application being executed, plus all the command-line arguments to this application. This option is useful for confirming options and files names passed to the compiler applications.

If this option is used twice (-V -V), it will display the full path to each compiler application as well as the full command line arguments. This would be useful to ensure that the correct compiler installation is being executed, if there is more than one compiler installed.

See **Section 2.8 "MPLAB IDE Universal Toolsuite Equivalents"** for use of this option in MPLAB IDE.

2.7.16   -X: Strip Local Symbols

The option -X strips local symbols from any files compiled, assembled or linked. Only global symbols will remain in any object files or symbol files produced. This option is not normally required for most projects.

2.7.17   --ASMLIST: Generate Assembler List Files

The `--ASMLIST` option tells PICC to generate *assembler listing files* for the C and assembly source modules being compiled. One assembly list file is produced for the entire C program, including code from the C library functions.

In addition, one assembly list file is produce for each assembly source file in the project, including the runtime startup code (see **Section 2.4.2 "Runtime Startup Code"**).

Assembly list files use a `.lst` extension and, due to the additional information placed in these files, cannot be used as assembly source files.

In the case of listings for C source code, the list file shows both the original C code and the corresponding assembly code generated by the compiler. See **Section 4.4 "Assembly List Files"** for full information regarding the content of these files.

The same information is shown in the list files for assembly source code.

This option is on by default when compiling in under MPLAB IDE and using the HI-TECH Universal Toolsuite.

2.7.18   --ADDRQUAL: Set Compiler Response to Memory Qualifiers

The --ADDRQUAL option indicates the compiler's response to non-standard memory qualifiers in C source code.

By default these qualifiers are ignored, i.e. they are accepted without error, but have no effect. Using this option allows these qualifiers to be binding and forces the compiler to position the qualified objects in the locations specified by the qualifiers. If the requirements of the qualifier cannot be met, an error will be generated.

The suboptions are detailed in Table 2-7.

TABLE 2-7:   COMPILER RESPONSES TO MEMORY QUALIFIERS

Selection	Response
`bank0, bank 1, bank2` or `bank3`	The `bank0 ... bank3` qualifiers are binding. See **Section 3.3.11.3 "Bank0, Bank1, Bank2 and Bank3 Type Qualifiers"**
`near`	The `near` qualifier is binding. See **Section 3.3.11.2 "Near Type Qualifier"**.
`eeprom`	The `eeprom` qualifier is binding. See **Section 3.3.11.4 "EEPROM Type Qualifier"**.

For example, the option:

```
--ADDRQUAL=bank0,near
```

requests that the compiler honour the `bank0` and `near` qualifiers, if present in C source code, but to ignore any other non-standard memory qualifiers encountered. If an object qualified as `bank0` cannot be placed into bank 0 data RAM, and error will be issued; if an object qualified `near` cannot be placed in the common memory, an error will also be issued.

2.7.19 --CHECKSUM: Calculate a checksum

This option will perform a checksum over the address range specified and store the result at the destination address specified. Additional specifications can be appended as a *comma*-separated list to this option. Such specifications are:

width=*n* selects the width of the checksum result in bytes. A negative width will store the result in little-endian byte order. Result widths from one to four bytes are permitted.

offset=*nnnn* specifies an initial value or offset to be added to this checksum.

algorithm=*n* select one of the checksum algorithms implemented in HEXMATE. The selectable algorithms are described in Table 6-9.

code=*nn* is a hexadecimal code that will trail each byte in the checksum result. This can allow each byte of the checksum result to be embedded within an instruction.

The *start*, *end* and *destination* attributes can be entered as word addresses as this is the native format for PICC program space. If an accompanying --FILL option has not been specified, unused locations within the specified address range will be filled with FFFh for Baseline devices, or 3FFFh for Mid-Range devices. This is to remove any unknown values from the equation and ensure the accuracy of the checksum result.

For example:

```
--checksum=800-fff@20,width=1,algorithm=8
```

will calculate a 1 byte checksum from address 0x800 to 0xfff and store this at address 0x20. Fletcher's algorithm will be used. See **Section 6.6.1.5 "-CK"**.

The checksum calculations are performed by the HEXMATE application. The information in this driver option is passed to the HEXMATE application when it is executed.

2.7.20 --CHIP: Define Processor

This option can be used to specify the target processor, or device, for the compilation. This is the only compiler option that is mandatory when compiling code.

To see a list of supported processors that can be used with this option, use the --CHIPINFO option described in **Section 2.7.21 "--CHIPINFO: Display List of Supported Processors"**.

2.7.21 --CHIPINFO: Display List of Supported Processors

The --CHIPINFO option displays a list of devices the compiler supports. The names listed are those chips defined in the chipinfo file and which may be used with the --CHIP option.

2.7.22 --CODEOFFSET: Offset Program Code to Address

In some circumstances, such as bootloaders, it is necessary to shift the program image to an alternative address. This option is used to specify a base address for the program code image and to reserve memory from address 0 to that specified in the option.

When using this option, all code psects (including reset and interrupt vectors and constant data) will be adjusted to the address specified. The address is assumed to be a hexadecimal constant. A leading **0X**, or a trailing h hexadecimal specifier can be used, but is not necessary.

This option differs to the --ROM option in that it will move the code associated with the reset and interrupt vectors which cannot be done using the --ROM option, see **Section 2.7.49 "--ROM: Adjust ROM Ranges"**.

For example, if the option `--CODEOFFSET=600` is specified, the reset vector will be moved from address 0 to address 600h; the interrupt vector will be moved from address 4 to 604h. No code will be placed between address 0 and 600h.

See **Section 2.8 "MPLAB IDE Universal Toolsuite Equivalents"** for use of this option in MPLAB IDE.

2.7.23   --CR: Generate Cross Reference Listing

The `--CR` option will produce a *cross reference listing*. If the `file` argument is omitted, the raw cross reference information will be left in a temporary cross reference file, leaving the user to run the CREF utility. If a filename is supplied, for example `--CR=test.crf`, PICC will invoke CREF to process the cross reference information into the listing file, in this case `test.crf`.

If multiple source files are to be included in the cross reference listing, all must be compiled and linked with the one PICC command. For example, to generate a cross reference listing which includes the source modules `main.c`, `module1.c` and `nvram.c`, compile and link using the command:

```
PICC --CHIP=16F877AA --CR=main.crf main.c module1.c nvram.c
```

Thus, this option cannot be used when using any compilation process that compiles each source file separately using the `-C` or `--PASS1` options. Such is the case for most IDEs, including MPLAB IDE, HI-TIDE, and make utilities.

See **Section 6.4 "Cref"** for information on the CREF utility.

2.7.24   --DEBUGGER: Select Debugger Type

This option is intended for use for compatibility with development tools which can act as a debugger. PICC supports several debuggers and using this option will configure the compiler to conform to the requirements of that selected. The possible selections for this option are defined in Table 2-8.

TABLE 2-8:   SELECTABLE DEBUGGERS

Suboption	Debugger selected
`none`	No debugger (default)
`icd` or `icd1`	MPLAB® ICD
`icd2`	MPLAB ICD 2
`icd3`	MPLAB ICD 3
`pickit2`	MPLAB PICkit™ 2
`pickit3`	MPLAB PICkit 3
`realice`	MPLAB REAL ICE™ in-circuit emulator

For example:

```
PICC --CHIP=16F877AA --DEBUGGER=icd2 main.c
```

Choosing the correct debugger is important as they can use memory resources which might be used by the project if this option is omitted. Such a conflict would lead to run-time failure.

If the debugging features of the development tool are not to be used, for example the MPLAB ICD 3 is only being used as a programmer, then the debugger option can be set to `none` as no memory resources will be used by the tool when operating in this way.

See **Section 2.8 "MPLAB IDE Universal Toolsuite Equivalents"** for use of this option in MPLAB IDE.

2.7.25 --DOUBLE: Select kind of Double Types

This option allows the kind of double precision floating point types to be selected. By default the compiler will choose the truncated IEEE754 24-bit implementation for `double` types. With this option, this can be changed to the full 32-bit IEEE754 implementation.

See **Section 2.8 "MPLAB IDE Universal Toolsuite Equivalents"** for use of this option in MPLAB IDE.

2.7.26 --ECHO: Echo command line before processing

Use of this option will result in the driver command line being echoed to the `stderr` stream before compilation commences. Each token of the command line will be printed on a separate line and will appear in the order in which they are placed on the command line.

2.7.27 --ERRFORMAT: Define Format for Compiler Messages

If the `--ERRFORMAT` option is not used, the default behavior of the compiler is to display any errors in a "human readable" form. This standard format is perfectly acceptable to a person reading the error output, but is not generally usable with environments which support compiler error handling.

This option allows the exact format of printed error messages to be specified using special placeholders embedded within a message template. See **Section 2.6 "Compiler Messages"** for full details of the messaging system employed by PICC, and the placeholders which can be used with this option.

This section is also applicable to the `--WARNFORMAT` and `--MSGFORMAT` options which adjust the format of warning and advisory messages, respectively.

2.7.28 --ERRORS: Maximum Number of Errors

This option sets the maximum number of errors each compiler application, as well as the driver, will display before execution is terminated. By default, up to 20 error messages will be displayed by each application.

See **Section 2.6 "Compiler Messages"** for full details of the messaging system employed by PICC.

2.7.29 --FILL: Fill Unused Program Memory

This option allows specification of a hexadecimal opcode that can be used to fill all unused program memory locations. Multi-byte codes should be entered in little endian byte order.

See **Section 2.8 "MPLAB IDE Universal Toolsuite Equivalents"** for use of this option in MPLAB IDE.

2.7.30   --FLOAT: Select kind of Float Types

This option allows the size of `float` types to be selected. The types available to be selected are given in Table 2-9.

See also the `--DOUBLE` option in **Section 2.7.25 "--DOUBLE: Select kind of Double Types"**.

TABLE 2-9:   FLOATING POINT SELECTIONS

Suboption	Effect
double	Size of float matches size of double type
24	24 bit float
32	32 bit float (IEEE754)

2.7.31   --GETOPTION: Get Command-line Options

This option is used to retrieve the command line options which are used for named compiler application. The options are then saved into the given file. This option is not required for most projects, and is disabled in Lite mode.

The options takes an application name and a filename to store the options, for example:

```
--GETOPTION=hlink,options.txt
```

2.7.32   --HELP: Display Help

This option displays information on the PICC compiler options. The option `--HELP` will display all options available. To find out more about a particular option, use the option's name as a parameter. For example:

```
PICC --help=warn
```

will display more detailed information about the `--WARN` option, the available suboptions, and which suboptions are enabled by default.

2.7.33   --IDE: Specify the IDE being used

This option is used to automatically configure the compiler for use by the named Integrated Development Environment (IDE). The supported IDEs are shown in Table 2-10.

TABLE 2-10:   SUPPORTED IDES

Suboption	IDE
hitide	HI-TECH's HI-TIDE™
mplab	Microchip's MPLAB® IDE

2.7.34   --LANG: Specify the Language for Messages

This option allows the compiler to be configured to produce error, warning and some advisory messages in languages other than English.

English is the default language unless this has been change at installation, or by the use of the `--SETUP` option. Some messages are only ever printed in English regardless of the language specified with this option. See **Section 2.6.2 "Message Language"** for more information.

Table 2-11 shows those languages currently supported.

TABLE 2-11:   SUPPORTED LANGUAGES

Suboption	Language
`en`, `english`	English
`fr`, `french`, `francais`	French
`de`, `german`, `deutsch`	German

2.7.35   --MEMMAP: Display Memory Map

This option will display a memory map for the specified map file. This option is seldom required, but would be useful if the linker is being driven explicitly, i.e. instead of in the normal way through the command-line driver. This command would display the memory summary which is normally produced at the end of compilation by the driver.

2.7.36   --MODE: Choose Compiler Operating Mode

This option selects the basic operating mode of the compiler. The available types are `pro`, `std` and `lite`. A compiler operating in PRO mode uses full optimization and produces the smallest code size. Standard mode uses limited optimizations, and LITE mode only uses a minimum optimization level and will produce relatively large code.

Only those modes permitted by the compiler license status will be accepted. For example if you have purchased a Standard compiler license, that compiler may be run in Standard mode, or lite mode, but not the PRO mode.

2.7.37   --MSGDISABLE: Disable Warning Messages

This option allows warning or advisory messages to be disabled during compilation of a project. The `messagelist` is a *comma*-separated list of warning numbers that are to be disabled. If the number of an error is specified, it will be ignored by this option. If the message list is specified as `0`, then all warnings are disabled. See **Section 2.6.5 "Changing Message Behavior"** for other ways to disable messages.

For full information on the compiler's messaging system, see **Section 2.6 "Compiler Messages"**.

2.7.38   --MSGFORMAT: Set Advisory Message Format

This option sets the format of advisory messages produced by the compiler. Warning and error messages are controlled separately by other options. See **Section 2.7.27 "--ERRFORMAT: Define Format for Compiler Messages"** and **Section 2.7.60 "--WARNFORMAT: Set Warning Message Format"** for information on change the format of these sorts of messages.

See **Section 2.6 "Compiler Messages"** for full information on the compiler's messaging system.

2.7.39   --NODEL: Do not remove temporary files

Specifying `--NODEL` when building will instruct PICC not to remove the intermediate and temporary files that were created during the build process.

2.7.40   --NOEXEC: Don't Execute Compiler

The `--NOEXEC` option causes the compiler to assemble all the command lines for the compiler applications, but not to perform any compilation or produce any output.

This may be useful when used in conjunction with the `-V` option (**Section 2.7.15 "-V: Verbose Compile"**) in order to see all of the command lines the compiler uses to drive the compiler applications.

2.7.41   --OBJDIR: Specify a directory for intermediate files

This option allows a directory to be nominated in for PICC to locate its intermediate files. If this option is omitted, intermediate files will be created in the current working directory.

This option will not set the location of output files, instead use --OUTDIR. See **Section 2.7.43 "--OUTDIR: Specify a directory for output files"** and **Section 2.7.10 "-O: Specify Output File"** for more information.

2.7.42   --OPT: Invoke Compiler Optimizations

The --OPT option allows control of all the compiler optimizers. If this option is not specified, or it is specified as --OPT=all, all optimizations are enabled. Optimizations may be disabled by using --OPT=none, or individual optimizers may be controlled, e.g. --OPT=asm will only enable some assembler optimizations.

Table 2-12 lists the available optimization types. The optimizations that are controlled through specifying a level 1 through 9 affect optimization during the code generation stage. The level selected is commonly referred to as the *global optimization level*.

TABLE 2-12:   OPTIMIZATION OPTIONS

Option name	Function
1...9	Select global optimization level (1 through 9)
asm	Select optimizations of assembly code derived from C source
asmfile	Select optimizations of assembly source files
debug	Favor accurate debugging over optimization
speed	Favor optimizations that result in faster code
space	Favor optimizations that result in smaller code
all	Enable all compiler optimizations
none	Do not use any compiler optimizations

Note that different suboptions control assembler optimizations of assembly source files and intermediate assembly files produced from C source code.

The speed and space suboptions are contradictory. Space optimizations are the default. If speed and space suboptions are both specified, then speed optimizations takes precedence. These optimizations affect procedural abstraction, which is performed by the assembler, and other optimizations at the code generation stage.

2.7.43   --OUTDIR: Specify a directory for output files

This option allows a directory to be nominated for PICC to locate its output files. If this option is omitted, output files will be created in the current working directory. See also **Section 2.7.41 "--OBJDIR: Specify a directory for intermediate files"** and **Section 2.7.10 "-O: Specify Output File"** for more information.

2.7.44   --OUTPUT= type: Specify Output File Type

This option allows the type of the output file(s) to be specified. If no --OUTPUT option is specified, the output file's name will be the same as the project name (see **Section 2.2.1 "Output Files"**).

The available output file format are shown in Table 2-13. More than one output format may be specified by supplying a *comma*-separated list of tags. Not all formats are supported by Microchip development tools.

Those output file types which specify library formats stop the compilation process before the final stages of compilation are executed. Hence specifying an output file format list containing, e.g. lib or all will prevent the other formats from being created.

TABLE 2-13:   OUTPUT FILE FORMATS

Type tag	File format
lib	(Object) Library File
lpp	P-code library
intel	Intel HEX
tek	Tektronic
aahex	American Automation symbolic HEX file
mot	Motorola S19 HEX file
ubrof	UBROF format
bin	Binary file
mcof	Microchip COFF
cof	Common Object File Format
cod	Bytecraft COD file format
elf	ELF/DWARF file format

2.7.45   --PASS1: Compile to P-code

The --PASS1 option is used to generate a p-code intermediate files (.p1 file) from the parser, then stop compilation. Such files need to be generated if creating a p-code library files, however the compiler is able to generate library files in one step, if required. See **Section 2.7.44 "--OUTPUT= type: Specify Output File Type"** for specifying a library output file type.)

2.7.46   --PRE: Produce Preprocessed Source Code

The --PRE option is used to generate preprocessed C source files (also called modules or translation units) with an extension .pre. This may be useful to ensure that preprocessor macros have expanded to what you think they should. Use of this option can also create C source files which do not require any separate header files. If the .pre files are renamed to .c files that can be passed to the compiler for subsequent processing. This is useful when sending files to a colleague or to obtain technical support without having to send all the header files, which may reside in many directories.

If you wish to see the preprocessed source for the printf() family of functions, do *not* use this option. The source for this function is customized by the compiler, but only after the code generator has scanned the project for printf() usage. Thus, as the --PRE option stops compilation after the preprocessor stage, the code generator will not execute and no printf() code will be processed. If this option is omitted, the preprocessed source for printf() will be automatically retained in the file doprnt.pre.

2.7.47   --PROTO: Generate Prototypes

The --PROTO option is used to generate .pro files containing both ANSI C and K&R style function declarations for all functions within the specified source files. Each .pro file produced will have the same base name as the corresponding source file. Prototype files contain both ANSI C-style prototypes and old-style C function declarations within conditional compilation blocks.

The extern declarations from each .pro file should be edited into a global header file which can then be included into all the C source files in the project. The .pro files may also contain static declarations for functions which are local to a source file. These static declarations should be edited into the start of the source file.

To demonstrate the operation of the `--PROTO` option, enter the following source code as file `test.c`:

```
#include <stdio.h>
add(arg1, arg2)
int *   arg1;
int *   arg2;
{
        return *arg1 + *arg2;
}


void printlist(int * list, int count)
{
        while (count--)
                printf("d " *list++);
        putchar('\n');
}
```

If compiled with the command:

```
PICC --CHIP=16F877AA --PROTO test.c
```

PICC will produce `test.pro` containing the following declarations which may then be edited as necessary:

```
/* Prototypes from test.c */
/* extern functions - include these in a header file */
#if     PROTOTYPES
extern int add(int *, int *);
extern void printlist(int *, int);
#else           /* PROTOTYPES */
extern int add();
extern void printlist();
#endif          /* PROTOTYPES */
```

2.7.48   --RAM: Adjust RAM Ranges

This option is used to adjust the default RAM which is specified for the target device. The default memory will include all the on-chip RAM specified for the target PIC10/12/16 device, thus this option only need be used if there are special memory requirements. Typically this option is used to reserve memory (reduce the amount of memory available). Specifying additional memory that is not in the target device will typically result in a successful compilation, but may lead to code failures at runtime.

The default RAM memory for each target device is specified in the chipinfo file, `PICC.INI`.

Strictly speaking, this option specifies the areas of memory that may be used by writable (RAM-based) objects, and not necessarily those areas of memory which contain physical RAM. The output that will be placed in the ranges specified by this option are typically variables that a program defines.

For example, to specify an additional range of memory to that present on-chip, use:

```
--RAM=default,+100-1ff
```

This will add the range from 100h to 1ffh to the on-chip memory. To only use an external range and ignore any on-chip memory, use:

```
--RAM=0-ff
```

This option may also be used to reserve memory ranges already defined as on-chip memory in the chipinfo file. To do this supply a range prefixed with a minus character, `-`, for example:

```
--RAM=default,-100-103
```

will use all the defined on-chip memory, but not use the addresses in the range from 100h to 103h for allocation of RAM objects.

This option will adjust the memory ranges used by linker classes, see **Section 5.2.1 "-Aclass =low-high,..."**, and hence any object which is in a psect placed in this class. Any objects which are contained in a psect that is explicitly placed at a memory address by the linker (see **Section 5.2.18 "-Pspec"**) i.e., are not placed into a memory class, are not affected by the option.

See **Section 2.8 "MPLAB IDE Universal Toolsuite Equivalents"** for use of this option in MPLAB IDE.

2.7.49   --ROM: Adjust ROM Ranges

This option is used to change the default ROM which is specified for the target device. The default memory will include all the on-chip ROM specified for the target PIC10/12/16 device, thus this option only need be used if there are special memory requirements. Typically this option is used to reserve memory (reduce the amount of memory available). Specifying additional memory that is not in the target device will typically result in a successful compilation, but may lead to code failures at runtime.

The default ROM memory for each target device is specified in the chipinfo file, `PICC.INI`.

Strictly speaking, this option specifies the areas of memory that may be used by read-only (ROM-based) objects, and not necessarily those areas of memory which contain physical ROM. When producing code that may be downloaded into a system via a bootloader the destination memory may indeed be some sort of (volatile) RAM. The output that will be placed in the ranges specified by this option are typically executable code and any data variables that are qualified as `const`.

For example, to specify an additional range of memory to that on-chip, use:

```
--ROM=default,+100-2ff
```

This will add the range from 100h to 2ffh to the on-chip memory. To only use an external range and ignore any on-chip memory, use:

```
--ROM=100-2ff
```

This option may also be used to reserve memory ranges already defined as on-chip memory in the chip configuration file. To do this supply a range prefixed with a minus character, `-`, for example:

```
--ROM=default,-100-1ff
```

will use all the defined on-chip memory, but not use the addresses in the range from 100h to 1ffh for allocation of ROM objects.

This option will adjust the memory ranges used by linker classes, see **Section 5.2.1 "-Aclass =low-high,..."**, and hence any object which is in a psect placed in this class. Any objects which are contained in a psect that is explicitly placed at a memory address by the linker (see **Section 5.2.18 "-Pspec"**) i.e., are not placed into a memory class, are not affected by the option.

See **Section 2.8 "MPLAB IDE Universal Toolsuite Equivalents"** for use of this option in MPLAB IDE.

2.7.50   --RUNTIME: Specify Runtime Environment

The --RUNTIME option is used to control what is included as part of the runtime environment. The runtime environment encapsulates any code that is present at runtime which has not been defined by the user, instead supplied by the compiler, typically as library code or compiler-generated source files.

All required runtime features are enabled by default and this option is not required for normal compilation. The usable suboptions include those shown in Table 2-14.

TABLE 2-14:   RUNTIME ENVIRONMENT SUBOPTIONS

Suboption	Controls	On (+) implies
init	The code present in the main program module that copies the ROM-image of initial values to RAM variables.	The ROM image is copied into RAM and initialized variables will contain their initial value at main().
clib	The inclusion of library files into the output code by the linker.	Library files are linked into the output.
clear	The code present in the main program module that clears uninitialized variables.	Uninitialized variables are cleared and will contain 0 at main().
download	Conditioning of the Intel HEX file for use with bootloaders.	Data records in the Intel HEX file are padded out to 16 byte lengths and will align on 16 byte boundaries. Startup code will not assume reset values in certain registers.
osccal	Initialize the oscillator with the oscillator constant.	Oscillator will be calibrated.
oscval:*value*	Set the internal clock oscillator calibration value.	Oscillator will be calibrated with *value* supplied.
keep	Whether the start-up module source file is deleted after compilation.	The start-up module and main program are not deleted.
resetbits	Preserve Power-down and Time-out STATUS bits at start up.	STATUS bits are preserved.
stackwarn	Checking the depth of the stack used.	The stack depth is monitored at compile time and a warning will be produced if a potential stack overflow is detected.
stackcall	Allow function calls to use the hardware stack.	Functions called via CALL instruction while stack not exhausted.

2.7.51   --SCANDEP: Scan for Dependencies

When this option is used, a .dep (dependency) file is generated. The dependency file lists those files on which the source file is dependant. Dependencies result when one file is #included into another.

2.7.52   --SERIAL: Store a Value at this Program Memory Address

This option allows a hexadecimal code to be stored at a particular address in program memory. A typical task for this option might be to position a serial number in program memory.

The byte-width of data to store is determined by the byte-width of the hexcode parameter in the option. For example, to store a one byte value, 0, at program memory address 1000h, use --SERIAL=00@1000. To store the same value as a four byte quantity use --SERIAL=00000000@1000.

This option is functionally identical to the corresponding HEXMATE option. For more detailed information and advanced controls that can be used with this option, refer to **Section 6.6.1.15 "-SERIAL"**.

The driver will also define a label at the location where the value was stored, and which can be referenced from C code as __serial0. To enable access to this symbol, remember to declare it, for example:

```
extern const int __serial0;
```

See **Section 2.8 "MPLAB IDE Universal Toolsuite Equivalents"** for use of this option in MPLAB IDE.

2.7.53   --SETOPTION: Set The Command-line Options for Application

This option is used to supply alternative command line options for the named application when compiling. The general form of this option is:

```
--SETOPTION=app,file
```

where the app component specifies the application that will receive the new options, and the file component specifies the name of the file that contains the additional options that will be passed to the application. This option is not required for most projects.

If specifying more than one option to a component, each option must be entered on a new line in the option file. This option can also be used to remove an application from the build sequence. If the *file* parameter is specified as *off*, execution of the named application will be skipped. In most cases this is not desirable as almost all applications are critical to the success of the build process. Disabling a critical application will result in catastrophic failure. However it is permissible to skip a non-critical application such as CLIST or HEXMATE if the final results are not reliant on their function.

2.7.54   --SHROUD:Obfuscate p-code Files

This options should be used in situations where either p-code files or p-code libraries are to be distributed and are built from confidential source code.

C comments, which are normally included into these files, as well as line numbers and variable names will be removed or obfuscated so that the original source code cannot be reconstructed from distributed files.

2.7.55   --STRICT: Strict ANSI Conformance

The --STRICT option is used to enable strict ANSI C conformance of all special, non-standard keywords.

HI-TECH C supports various special keywords (for example the persistent type qualifier). If the --STRICT option is used, these keywords are changed to include two *underscore* characters at the beginning of the keyword (e.g. __persistent) so as to strictly conform to the ANSI standard. Thus if you use this option, you will need to use the qualifier __persistent in your code, not persistent.

Be warned that use of this option may cause problems with some standard header files (e.g. <htc.h>) as they contain special keywords.

2.7.56   --SUMMARY: Select Memory Summary Output Type

Use this option to select the type of memory summary that is displayed after compilation. By default, or if the mem suboption is selected, a memory summary is shown. This shows the total memory usage for all memory spaces.

A psect summary may be shown by enabling the `psect` suboption. This shows individual psects, after they have been grouped by the linker, and the memory ranges they cover. Table 2-15 shows what summary types are available.

TABLE 2-15:   MEMORY SUMMARY SUBOPTIONS

Suboption	Controls	On (+) implies
psect	Summary of psect usage.	A summary of psect names and the addresses they were linked at will be shown.
mem	General summary of memory used.	A concise summary of memory used will be shown.
class	Summary of class usage.	A summary of all classes in each memory space will be shown.
hex	Summary of address used within the HEX file.	A summary of addresses and HEX files which make up the final output file will be shown.
file	Whether summary information is shown on the screen or shown and saved to a file.	Summary information will be shown on screen and saved to a file.

See **Section 2.8 "MPLAB IDE Universal Toolsuite Equivalents"** for use of this option in MPLAB IDE.

2.7.57   --TIME: Report time taken for each phase of build process

Adding the `--TIME` option when building generate a summary which shows how much time each stage of the build process took to complete.

2.7.58   --VER: Display The Compiler's Version Information

The `--VER` option will display what version of the compiler is running and then exit the compiler.

2.7.59   --WARN: Set Warning Level

The `--WARN` option is used to set the compiler warning level threshold. Allowable warning levels range from -9 to 9. The warning level determines how pedantic the compiler is about dubious type conversions and constructs. The higher the warning level, the more important the warning message. The default warning level is `0` and will allow all normal warning messages.

Use this option with care as some warning messages indicate code that is likely to fail during execution, or compromise portability.

Warning message can be individually disabled with the `--MSGDISABLE` option, see **Section 2.7.37 "--MSGDISABLE: Disable Warning Messages"**. See also **Section 2.6 "Compiler Messages"** for full information on the compiler's messaging system.

See **Section 2.8 "MPLAB IDE Universal Toolsuite Equivalents"** for use of this option in MPLAB IDE.

2.7.60   --WARNFORMAT: Set Warning Message Format

This option sets the format of warning messages produced by the compiler. See **Section 2.7.27 "--ERRFORMAT: Define Format for Compiler Messages"** for more information on this option. For full information on the compiler's messaging system, see **Section 2.6 "Compiler Messages"**.

2.8 MPLAB IDE UNIVERSAL TOOLSUITE EQUIVALENTS

When compiling from within Microchip's MPLAB IDE, it is still the compiler's command-line driver, PICC, that is being executed and compiling the program. The HI-TECH Universal Toolsuite plugin manages the MPLAB IDE _Build Options_ dialog that is used to access the compiler options, and most of these graphical controls ultimately adjust the driver's command-line options. You can see the command-line options being used when building in the _Output_ window in MPLAB IDE.

The following dialogs and descriptions identify the mapping between the dialog controls and command-line options. As the toolsuite is universal across all HI-TECH compilers, not all options are applicable for HI-TECH C Compiler for PIC10/12/16 MCUs.

2.8.1 Directories Tab

The options in this dialog control the output and search directories for some files. See Figure 2-5 in conjunction with the following command line option equivalents.

FIGURE 2-5: THE DIRECTORIES TAB
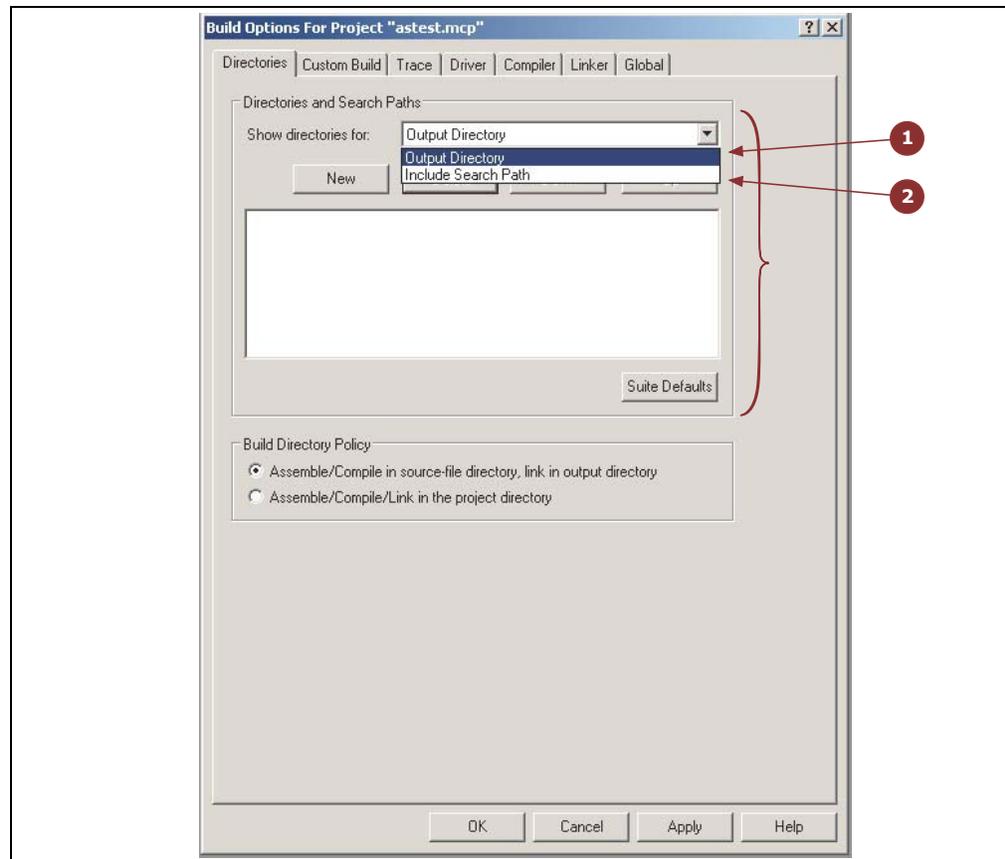
1. The output directory: This selection uses the buttons and fields grouped in the bracket to specify an output directory for files output by the compiler. This selection is handled internally by MPLAB IDE and does not use a driver option, however it is functionally equivalent to the --OUTDIR driver option (see **Section 2.7.43 "--OUTDIR: Specify a directory for output files"**).
2. Include Search path: This selection uses the buttons and fields grouped in the bracket to specify include (header) file search directories. See **Section 2.7.5 "-I: Include Search Path"**.

2.8.2 Compiler Tab

The options in this dialog control the aspects of compilation up to code generation. See Figure 2-6 in conjunction with the following command line option equivalents.

1. Define macros: The buttons and fields grouped in the bracket can be used to define preprocessor macros. See **Section 2.7.2 "-D: Define Macro"**.
2. Undefine macros: The buttons and fields grouped in the bracket can be used to undefine preprocessor macros. See **Section 2.7.14 "-U: Undefine a Macro"**.
3. Preprocess assembly: This checkbox controls whether assembly source files are scanned by the preprocessor. See **Section 2.7.11 "-P: Preprocess Assembly Files"**.

FIGURE 2-6: THE COMPILER TAB
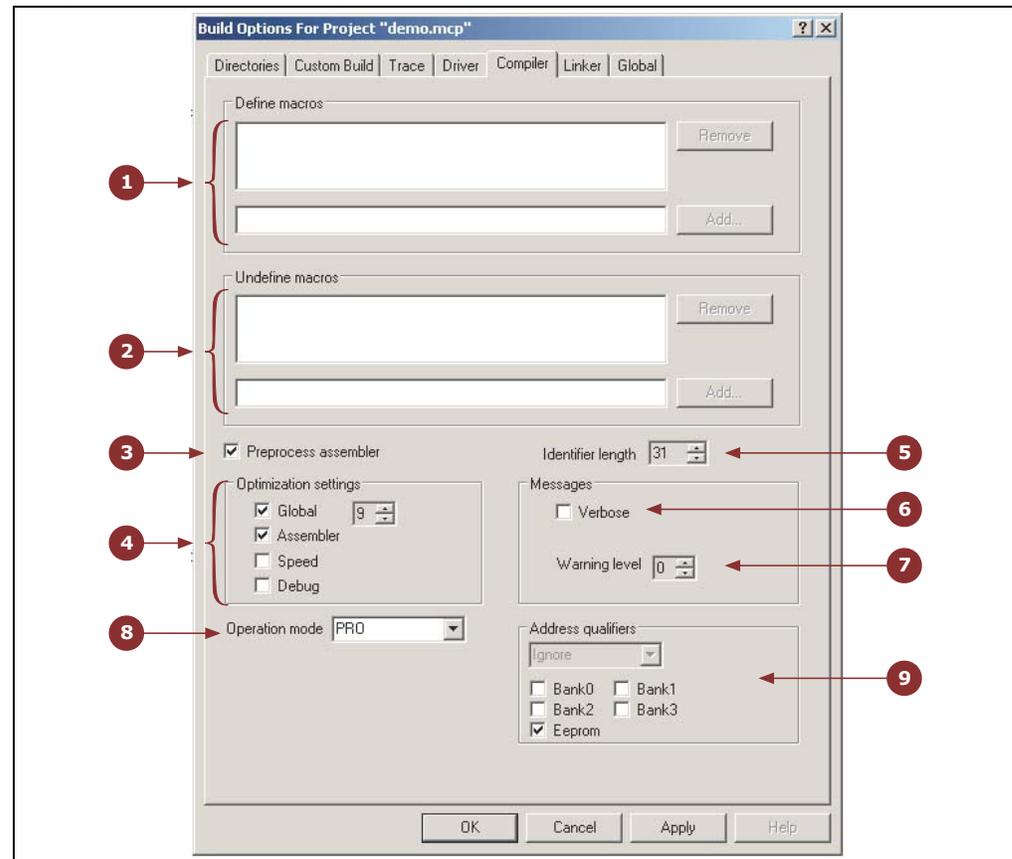
4. Optimization settings: These controls are used to adjust the different optimizations the compiler employs. See **Section 2.7.42 "--OPT: Invoke Compiler Optimizations"**.
5. Identifier length: This selector controls the maximum identifier length in C source. See **Section 2.7.9 "-N: Identifier Length"**.
6. Verbose: This checkbox controls whether the full command-lines for the compiler applications are displayed when building. See **Section 2.7.15 "-V: Verbose Compile"**.
7. Warning level: This selector allows the warning level print threshold to be set. See **Section 2.7.59 "--WARN: Set Warning Level"**.

8. Operation Mode: This selector allows the user to force another available operating mode (e.g. Lite, Standard or PRO) other then the default. See **Section 2.7.36 "--MODE: Choose Compiler Operating Mode"**

9. Address Qualifier: This selector allows the user to select the behavior of the address qualifier. See **Section 2.7.18 "--ADDRQUAL: Set Compiler Response to Memory Qualifiers"**

2.8.3   Linker Tab

The options in this dialog control the link step of compilation. See Figure 2-7 in conjunction with the following command line option equivalents.

FIGURE 2-7:   THE LINKER TAB
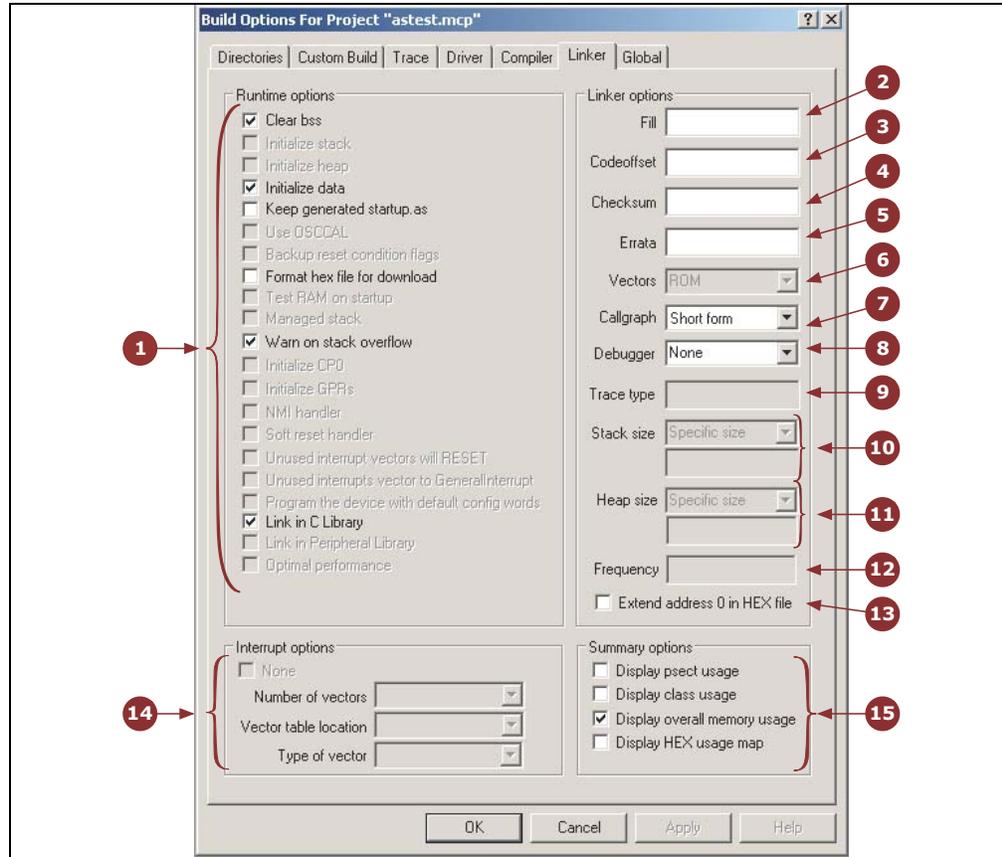
1. Runtime options: These checkboxes control the many runtime features the compiler can employ. See **Section 2.7.50 "--RUNTIME: Specify Runtime Environment"**.

2. Fill: This field allows a fill value to be specified for unused memory locations. See **Section 2.7.29 "--FILL: Fill Unused Program Memory"**.

3. Codeoffset: This field allows an offset for the program to be specified. See **Section 2.7.22 "--CODEOFFSET: Offset Program Code to Address"**.

4. Checksum: This field allows the checksum specification to be specified. See **Section 2.7.19 "--CHECKSUM: Calculate a checksum"**.

5. Errata: Not applicable.

6. Vectors: Not applicable.

7. Callgraph: Not applicable.
8. Debugger: This selector allows the type of hardware debugger to be chosen. See **Section 2.7.24 "--DEBUGGER: Select Debugger Type"**.
9. Trace type: Not yet implemented.
10. Stack size: Not applicable.
11. Heap size: Not applicable.
12. Frequency: Not applicable.
13. Extend address 0 in HEX file: This option specifies that the intel HEX file should have initialization to zero of the upper address. See **Section 2.7.44 "--OUTPUT= type: Specify Output File Type"**.
14. Interrupt options: Not applicable.
15. Summary Options: These checkboxes control which summaries are printed after compilation. See **Section 2.7.56 "--SUMMARY: Select Memory Summary Output Type"**.

2.8.4 Global Tab

The options in this dialog control aspects of compilation that are applicable throughout code generation and link steps — the second stage of compilation. See Figure 2-8 in conjunction with the following command line option equivalents.

FIGURE 2-8:   THE GLOBAL TAB
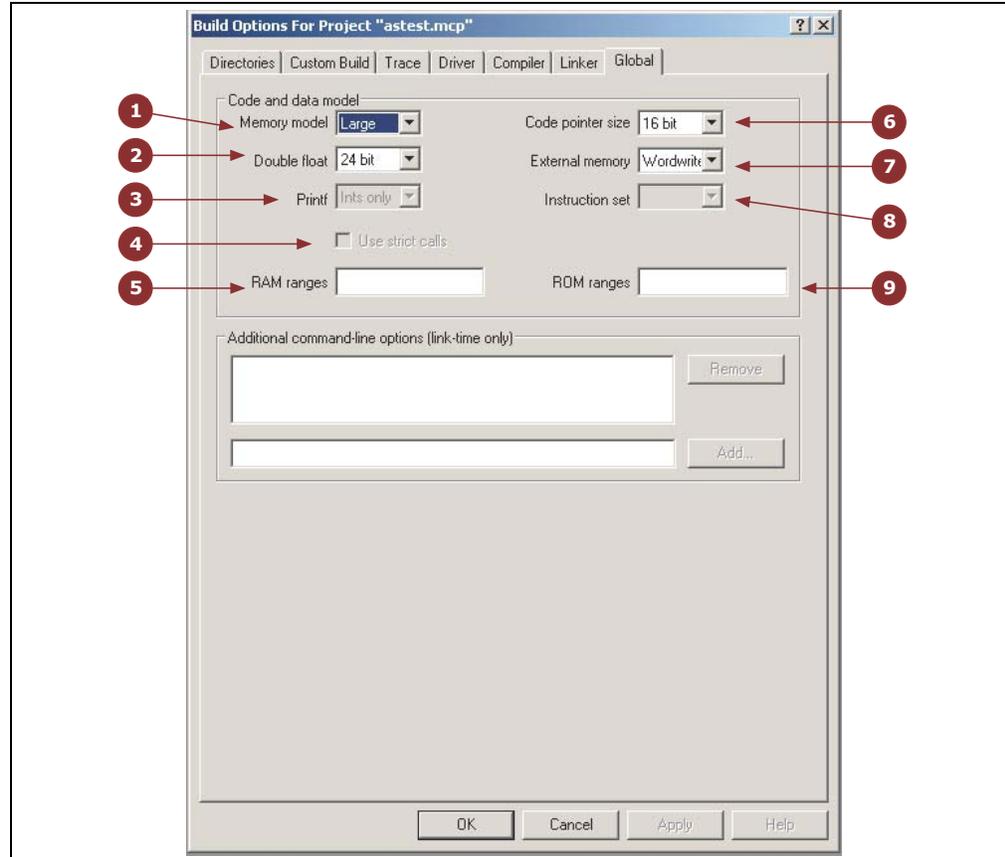
1. Memory model: Not applicable.
2. Double float: This selector allows the size of double float objects to be selected. See **Section 2.7.25 "--DOUBLE: Select kind of Double Types"**.
3. Printf: Not applicable.
4. Use strict calls: Not applicable.
5. RAM ranges: This field allows the default RAM (data space) memory used to be adjusted. See **Section 2.7.48 "--RAM: Adjust RAM Ranges"**.
6. Code pointer size: Not applicable.
7. External memory: Not applicable.
8. Instruction set: Not applicable.
9. ROM ranges: This field allows the default ROM (program space) memory used to be adjusted. See **Section 2.7.49 "--ROM: Adjust ROM Ranges"**.

Chapter 3. C Language Features

HI-TECH C Compiler for PIC10/12/16 MCUs supports a number of special features and extensions to the C language which are designed to ease the task of producing ROM-based applications. This chapter documents the special language features which are specific to these devices.

3.1   ANSI STANDARD ISSUES

3.1.1   Implementation-defined behavior

Certain sections of the ANSI C standard have implementation-defined behavior. This means that the exact behavior of some C code can vary from compiler to compiler. Throughout this manual are sections describing how the HI-TECH C compiler behaves in such situations.

3.2   PROCESSOR-RELATED FEATURES

HI-TECH C has several features which relate directly to the PIC10/12/16 architectures and instruction sets. These detailed in the following sections.

3.2.1   Stack

The hardware stack on PIC devices is limited in depth and cannot be manipulated directly. It is only used for function return address and cannot be used for program data. The compiler implements a compiled stack for local data objects, see **Section 3.4.2 "Compiled Stack Operation"** for information on how this is achieved.

You must ensure that the maximum stack depth is not exceeded otherwise code may fail.

A call graph is provided by the code generator in the assembler list file. This will indicate the stack levels at each function call and can be used as a guide to stack depth. The code generator may also produce warnings if the maximum stack depth is exceeded.

Both of these are guides to stack usage. Optimizations and the use of interrupts can decrease or increase, respectively, the stack depth used by a program over that determined by the compiler.

3.2.2   Configuration Fuses

The PIC device processor's configuration fuses (or configuration bits) may be set using the `__CONFIG()` macro as follows:

```
__CONFIG(x);
```

Note there are two leading *underscore* characters. The macro is defined in `<htc.h>` header file, so be sure to include this into the files that uses this macro.

The *x* argument is the value that is to be programmed in the configuration word. The value can either be a literal or be built up from specially named quantities are defined in the header file appropriate for the processor you are using. These macro names are similar to the names as used in the PIC10/12/16 data sheets to represent the configuration conditions and must be bitwise ANDed together to form the configuration value. Refer to your processor's header file for details. For example:

```
#include <htc.h>
__CONFIG(WDTDIS & HS & UNPROTECT);
```

For devices that have more than one configuration word location, each subsequent invocation of `__CONFIG()` will modify the next configuration word in sequence. Typically this might look like:

```
#include <htc.h>
__CONFIG(WDTDIS & XT & UNPROTECT);  // Program config. word 1
__CONFIG(FCMEN);                    // Program config. word 2
```

3.2.3   ID Locations

Some PIC10/12/16 devices have locations outside the addressable memory area that can be used for storing program information, such as an ID number. The `__IDLOC()` macro may be used to place data into these locations. The macro is used in a manner similar to:

```
#include <htc.h>
__IDLOC(x);
```

where *x* is a list of nibbles which are to be positioned into the ID locations. Only the lower four bits of each ID location is programmed, so the following:

```
__IDLOC(15F0);
```

will attempt to fill ID locations with the values: 1, 5, F and 0.

The base address of the ID locations is specified by the `idloc` psect which will be automatically assigned as appropriate address based on the type of device selected.

Some devices will permit programming up to seven bits within each ID location. To program the full seven bits, the regular `__IDLOC()` macro is not suitable. For this situation the `__IDLOC7(a,b,c,d)` macro is available. The parameters *a* to *d* are the values to be programmed. The values can be entered in either decimal or hexadecimal format, such as:

```
__IDLOC7(0x7f,1,70,0x5a);
```

It is not appropriate to use the `__IDLOC7()` macro on a device that does not permit seven bit programming of ID locations.

3.2.4   Bit Instructions

Wherever possible, HI-TECH C will attempt to use the PIC10/12/16 bit instructions. For example, when using a bitwise operator and a mask to alter a bit within an integral type, the compiler will check the mask value to determine if a bit instruction can achieve the same functionality.

```
unsigned int foo;
foo |= 0x40;
```

will produce the instruction:

```
BSF _foo,6
```

To set or clear individual bits within integral type, the following macros could be used:

```
#define bitset(var, bitno)    ((var) |= 1UL << (bitno))
#define bitclr(var, bitno)    ((var) &= ~(1UL << (bitno)))
```

To perform the same operation as above, the `bitset` macro could be employed as follows:

```
bitset(foo,6);
```

3.2.5   EEPROM Access

For most devices that come with on-chip EEPROM, the compiler offers several methods of accessing this memory. The EEPROM access methods are described in the following sections.

3.2.5.1   THE __EEPROM_DATA() MACRO

For those PIC10/12/16 devices that support external programming of their EEPROM data area, the `__EEPROM_DATA()` macro can be used to place the initial EEPROM data values into the HEX file ready for programming. The macro is used as follows.

```
#include <htc.h>
__EEPROM_DATA(0, 1, 2, 3, 4, 5, 6, 7);
```

The macro accepts eight parameters, being eight data values. Each value should be a byte in size. Unused values should be specified as a parameter of zero.

The macro may be called multiple times to define the required amount of EEPROM data. It is recommended that the macro be placed outside any function definitions.

The macro defines, and places the data within, a psect called `eeprom_data`. This psect is automatically positioned by the linker.

This macro is not used to write to EEPROM locations during runtime, it is to be used for pre-loading EEPROM contents at program time only.

3.2.5.2 EEPROM ACCESS FUNCTIONS

The library functions `eeprom_read()` and `eeprom_write()`, can be called to read from, and write to, the EEPROM during program execution. For example, to write a byte-size value to an address in EEPROM and retrieve it using these functions would be:

```
#include <htc.h>
void eetest(void) {
    unsigned char value = 1;
    unsigned char address = 0;

    // write value to EEPROM address
    eeprom_write(address, value);
    // read from EEPROM at address
    value = eeprom_read(address);
}
```

These functions test and wait for any concurrent writes to EEPROM to conclude before performing the required operation. The `eeprom_write()` function will initiate the process of writing to EEPROM and this process will not have completed by the time that `eeprom_write()` returns. The new data written to EEPROM will become valid approximately four milliseconds later.

In the above example, the new value will not yet be ready at the time when `eeprom_read()` is called, however because this function waits for any concurrent writes to complete before initiating the read, the correct value will be read.

It may also be convenient to use the preprocessor symbol, `_EEPROMSIZE` in conjunction with some of these access methods. This symbol defines the number of EEPROM bytes available for the selected chip.

3.2.5.3 EEPROM ACCESS MACROS

Although these macros perform much the same service as their library function counterparts, these should only be employed in specific circumstances. It is appropriate to select `EEPROM_READ` or `EEPROM_WRITE` in favor of the library equivalents if any of the following conditions are true:

- You cannot afford the extra level of stack depth required to make a function call
- You cannot afford the added code overhead to pass parameters and perform a call/return
- You cannot afford the added processor cycles to execute the function call overhead

Be aware that if a program contains multiple instances of either macro, any code space saving will be negated as the full content of the macro is now duplicated in code space.

In the case of `EEPROM_READ()`, there is another very important detail to note. Unlike `eeprom_read()`, this macro does not wait for any concurrent EEPROM writes to complete before proceeding to select and read EEPROM. Had the previous example used the `EEPROM_READ()` macro in place of `eeprom_read()` the operation would have failed. If it cannot be guaranteed that all writes to EEPROM have completed at the time of calling `EEPROM_READ()`, the appropriate flag should be polled prior to executing `EEPROM_READ()`.

For example:

```c
#include <htc.h>
void eetest(void){
    unsigned char value = 1;
    unsigned char address = 0;

    // Initiate writing value  to address
    EEPROM_WRITE(address,value);
    // wait for end-of-write before EEPROM_READ
    while(WR)
        continue;    // read from EEPROM at address
    value = EEPROM_READ(address);
}
```

3.2.5.4 EEPROM QUALIFIER

Variables may be directly qualified as `eeprom`. This places them in EEPROM memory of the device. For example:

```c
eeprom long prod_ID;
```

This qualifier allows individual named variables to be defined and accessed in the C code. Although this is a convenient way to define EEPROM-based objects and access them in C code, access will be much slower compared to reading and writing RAM-based variables.

3.2.6 Flash Runtime Access

HI-TECH C Compiler for PIC10/12/16 MCUs provides a number of methods to access the contents of program memory at runtime.

Particular care must be taken when modifying the contents of program memory. If the location being modified is that of the code that is currently being executed, or a region of your executable code has been used for use as non-volatile storage, the code may fail.

For those devices requiring a Flash erasure operation be performed prior to writing to Flash, this step will be performed internally by the compiler within the access routine and does not need to be implemented as a separate stage. Data within the same Flash erasure block that is unrelated to the write operation will be backed up before the block is erased and restored after the erasure.

3.2.6.1 FLASH ACCESS MACROS

Similar to the EEPROM read/write routines described above, there are equivalent Flash memory routines. For example, to write a byte-sized value to an address in Flash memory use:

```c
FLASH_WRITE(address,value);
```

To read a byte of data from an address in Flash memory, and store it in a variable, use:

```c
variable=FLASH_READ(address);
```

3.2.6.2 FLASH ACCESS FUNCTIONS

The `flash_read()` function provides the same functionality as the `FLASH_READ()` macro but will potentially cost less in code space if multiple invocations are required.

The `flash_copy()` function allows duplication of a block of memory at a location in Flash memory. The block of data being duplicated can be sourced from either RAM or program memory. This routine is only available for those devices which support writes to Flash memory in sizes greater than one word.

For the small subset of devices which allow independent control over a Flash block erasure process, the `flash_erase()` function provides this service. It is not required for other devices.

3.2.7 Baseline PIC MCU special instructions

The baseline (12-bit instruction word) devices have some registers which are not in the normal SFR area and cannot be accessed using an ordinary file instruction. The HI-TECH C compiler is instructed to automatically use the special instructions intended for such cases when pre-defined symbols are accessed.

The definition of the special symbols make use of the `control` qualifier. This qualifier informs the compiler that the registers are outside of the normal address space and that a different access method is required.

3.2.7.1 THE OPTION INSTRUCTION

Some baseline PIC devices use an `OPTION` instruction to load the `OPTION` register. The `<htc.h>` header file will ensure a special definition for a C object called `OPTION` and macros for the bit symbols which are stored in this register. PICC will automatically use the `OPTION` instruction when an appropriate processor is selected and the `OPTION` register is accessed.

For example, to set the prescaler assignment bit so that prescaler is assigned to the watchdog timer, the following code can be used.

```
OPTION = PSA;
```

This will load the appropriate value into the `W` register and then call the `OPTION` instruction.

3.2.7.2 THE TRIS INSTRUCTIONS

Some PIC devices use a `TRIS` instruction to load the `TRIS` register. The `<htc.h>` header file will ensure a special definition for a C object called `TRIS`. PICC will automatically use the `TRIS` instruction when an appropriate processor is selected and the `TRIS` register is accessed.

For example, to make all the bits on the output port high impedance, the following code can be used.

```
TRIS = 0xFF;
```

This will load the appropriate value into the `W` register and then call the `TRIS` instruction.

Those PIC devices which have more than one output port may have definitions for objects: `TRISA`, `TRISB` and `TRISC`, depending on the exact number of ports available. This objects are used in the same manner as described above.

3.2.7.3 OSCILLATOR CALIBRATION CONSTANTS

Some PIC devices come with an oscillator calibration constant which is pre-programmed into the devices program memory. This constant can be read from program memory and written to the `OSCCAL` register to calibrate the internal RC oscillator.

On some baseline PIC devices, the calibration constant is stored as a `MOVLW` instruction at the top of program memory, e.g. the PIC12C50X and PIC16C505 parts. On Reset, the program counter is made to point to this instruction and it is executed first before the program counter wraps around to 0x0000, which is the effective reset vector for the device. The default HI-TECH C startup routine will automatically include code to load the `OSCCAL` register with the value contained in the `W` register after reset on such devices. No other code is required.

For other chips, such as PIC12C67X chips, the oscillator constant is also stored at the top of program memory, but as a `RETLW` instruction. The compiler's startup code will automatically generate code to retrieve this value and perform the configuration.

Loading of the calibration value can be turned off via the `--RUNTIME` option (see **Section 2.7.50 "--RUNTIME: Specify Runtime Environment"**).

At runtime this calibration value may be read using the macro `_READ_OSCCAL_DATA()`. To be able to use this macro, make sure that `<htc.h>` is included into the relevant modules of your program. This macro returns the calibration constant which can then be stored into the `OSCCAL` register, as follows:

```
OSCCAL = _READ_OSCCAL_DATA();
```

> **Note:** The location which stores the calibration constant is never code protected and will be lost if you reprogram the device. Thus, if you are using a windowed or Flash device, the calibration constant must be saved from the last ROM location before it is erased. The constant must then be reprogrammed at the same location along with the new program and data.
>
> If you are using an in-circuit emulator (ICE), the location used by the calibration `RETLW` instruction may not be programmed. Calling the `_READ_OSCCAL_DATA()` macro will not work and will almost certainly not return correctly. If you wish to test code that includes this macro on an ICE, you will have to program a `RETLW` instruction at the appropriate location in program memory. Remember to remove this instruction when programming the actual part so you do not destroy the calibration value.

3.3   SUPPORTED DATA TYPES AND VARIABLES

The HI-TECH C compiler supports basic data types with 1, 2, 3 and 4 byte sizes. Table 3-1 shows the data types and their corresponding size and arithmetic type. The default type for each type is underlined. These types are further described in **Section 3.3.3 "Bit Data Types and Variables"** through **Section 3.3.8 "Floating Point Types and Variables"**.

TABLE 3-1:   BASIC DATA TYPES

Type	Size (bits)	Arithmetic Type
bit	1	Unsigned integer
signed char	8	Signed integer
unsigned char	8	Unsigned integer
signed short	16	Signed integer
unsigned short	16	Unsigned integer
signed int	16	Signed integer
unsigned int	16	Unsigned integer
signed short long	24	Signed integer
unsigned short long	24	Unsigned integer
signed long	32	Signed integer
unsigned long	32	Unsigned integer
float	24	Real
double	24 or 32	Real

3.3.1   Object Size

Arrays of any type (including arrays of aggregate types) are fully supported. So too are the structure and union aggregate types, see **3.3.9 "Structures and Unions"**.

When compiling for enhanced Mid-Range PIC devices, the size of an object (array or aggregate object) is typically limited only by the total available data memory. Single objects which will not fit into any of the available general purpose RAM ranges will be allocated memory in several RAM banks. The memory locations taken up by such objects would normally be considered as non-contiguous for any assembly code which uses direct addressing, but can be arranged using the device's linear GPR memory so that indirect accesses will see a contiguous memory block.

Note that the special function registers (which reside in the data memory space) or memory reservations in general purpose RAM may prevent objects from being allocated contiguous memory in the one bank and so objects that are smaller than the size of a RAM bank may be allocated across multi-banks. The generated code to access multi-bank objects will always be accessed indirectly, be slower and the associated code size will be large than for objects fully contained within a single RAM bank.

On baseline and other mid-range devices, arrays are limited to the maximum size of the available general purpose memory in each RAM bank. An error will result if an array is defined which is larger than this size.

3.3.2   Radix Specifiers and Constants

The format of integral constants specifies their radix. HI-TECH C supports the ANSI standard radix specifiers as well as ones which enables binary constants to be specified in C code.

The format used to specify the radices are given in Table 3-2. The letters used to specify binary or hexadecimal radices are case insensitive, as are the letters used to specify the hexadecimal digits.

TABLE 3-2:   RADIX FORMATS

Radix	Format	Example
binary	`0b` *number* or `0B` *number*	0b10011010
octal	`0` *number*	0763
decimal	*number*	129
hexadecimal	`0x` *number* or `0X` *number*	0x2F

Any integral constant will have a type of `int`, or a larger type, such that the type can hold the value without overflow; however, later optimizations may reduce this to a smaller type. The suffix `l` or `L` may be used with the constant to indicate that it must be assigned either a `signed long int` or `unsigned long int` type, and the suffix `u` or `U` may be used with the constant to indicate that it must be assigned an unsigned type, and both `l` or `L` and `u` or `U` may be used to indicate `unsigned long int` type.

Floating-point constants have `double` type unless suffixed by `f` or `F`, in which case it is a `float` constant. The suffixes `l` or `L` specify a `long double` type which is considered an identical type to `double` by `HI-TECH C`.

Character constants are enclosed by single quote characters, `'`, for example `'a'`. A character constant has `char` type.

Multi-byte character constants are not supported.

String constants or string literals are enclosed by double quote characters `"`, for example `"hello world"`. The type of string constants is `const char *` and the character that make up the string are stored in the program memory as are all objects qualified `const`.

Assigning a string literal to a pointer to a non-`const char` will generate a warning from the compiler. This code is legal, but the behavior if the pointer attempts to write to the string will fail. For example:

```
char * cp= "one";        // "one"  in ROM, produces warning
const char * ccp= "two"; // "two"  in ROM, correct
```

Defining and initializing a non-`const` array (i.e. not a pointer definition) with a string, for example:

```
char ca[]= "two";        // "two"  different to the above
```

is a special case and produces an array in data space which is initialized at startup with the string `"two"` (copied from program space), whereas a string constant used in other contexts represents an unnamed `const` -qualified array, accessed directly in program space.

HI-TECH C will use the same storage location and label for strings that have identical character sequences, except where the strings are used to initialize an array residing in the data space as shown in the last statement in the previous example. For example, in the code snippet

```
if(strncmp(scp, "hello", 6) == 0)
    fred = 0;
if(strcmp(scp, "world") == 0)
    fred--;
if(strcmp(scp, "hello world") == 0)
    fred++;
```

the characters in the string `"world"` and the last 6 characters of the string `"hello world"` (the last character is the nul terminator character) would be represented by the same `RETLW` instructions stored at the same memory locations. The string `"hello"` would not overlap with the same characters in the string `"hello world"` as they differ in terms of the placement of the nul character.

Two adjacent string constants (i.e. two strings separated *only* by white space) are concatenated by the compiler. Thus:

```
const char * cp = "hello"  "world";
```

will assign the pointer with the address of the string `"hello world"`.

3.3.3   Bit Data Types and Variables

HI-TECH C Compiler for PIC10/12/16 MCUs supports `bit` integral types which can hold the values 0 or 1. Single `bit` variables, or booleans, may be declared using the keyword `bit`, for example:

```
bit init_flag;
```

Boolean variables cannot be `auto` or parameters to a function, but can be qualified `static`, allowing them to be defined locally within a function. For example:

```
int func(void) {
    static bit flame_on;
    // ...
}
```

A function may return a `bit` object by using the `bit` keyword in the function's prototype in the usual way. The boolean will be returned in the carry flag in the `STATUS` register.

The `bit` variables behave in most respects like normal `unsigned char` variables, but they may only contain the values 0 and 1, and therefore provide a convenient and efficient method of storing boolean flags. Eight boolean objects are packed into each byte of memory storage, so they don't consume large amounts of internal RAM.

Operations on `bit` objects are performed using the single bit instructions (`bsf` and `bcf`) wherever possible, thus the generated code to access `bit` objects is very efficient.

It is not possible to declared a pointer to `bit` types or assign the address of a `bit` object to any pointer. Nor is it possible to statically initialise `bit` variables so they must be assigned any non-zero starting value (i.e. a true value) in the code itself. Bit objects will be cleared on startup, unless the bit is qualified persistent.

When assigning a larger integral type to a `bit` variable, only the Least Significant bit is used. For example, if the `bit` variable `bitvar` was assigned as in the following:

```
int data = 0x54;
bit bitvar;
bitvar = data;
```

it will be cleared by the assignment since the Least Significant bit of `data` is zero. If you want to set a bit variable to be 0 or 1 depending on whether the larger integral type is zero (false) or non-zero (true), use the form:

```
bitvar = (data != 0);
```

The psects in which `bit` objects are allocated storage are declared using the `bit` `PSECT` directive flag, see **Section 4.3.9.3 "PSECT"**. All addresses specified for bit objects and psects will be bit addresses. Take care when comparing these addresses to byte addresses used by all other variables.

If the PICC flag `--STRICT` is used, the `bit` keyword becomes unavailable.

3.3.4   8-Bit Integer Data Types and Variables

HI-TECH C Compiler for PIC10/12/16 MCUs supports both `signed char` and `unsigned char` 8-bit integral types. If the `signed` or `unsigned` keyword is absent from the variable's definition, the default type is `unsigned char`.

The `signed char` type is an 8-bit two's complement signed integer type, representing integral values from -128 to +127 inclusive. An `unsigned char` is an 8-bit unsigned integer type, representing integral values from 0 to 255 inclusive.

It is a common misconception that the C `char` types are intended purely for ASCII character manipulation. This is not true, indeed the C language makes no guarantee that the default character representation is even ASCII. The `char` types are simply the smallest of up to four possible integer sizes, and behave in all respects like integers. The reason for the name "char" is historical and does not mean that `char` can only be used to represent characters. It is possible to freely mix `char` values with `short`, `int` and `long` values in C expressions. With HI-TECH C the `char` types will commonly be used for a number of purposes, as 8-bit integers, as storage for ASCII characters, and for access to I/O locations.

3.3.5   16-Bit Integer Data Types

HI-TECH C Compiler for PIC10/12/16 MCUs supports four 16-bit integer types. The types `short` and `int` hold 16-bit two's complement signed integers, representing integral values from -32,768 to +32,767 inclusive. They types `unsigned short` and `unsigned int` hold 16-bit unsigned integers, representing integral values from 0 to 65,535 inclusive.

All 16-bit integer values are represented in little endian format with the Least Significant Byte at the lower address.

Variables may be declared using the `signed short int` or `signed int` and `unsigned short int` or `unsigned int` keyword sequences, respectively, to hold values of these types. When specifying a `short int` type, the keyword `int` may be omitted. Thus a variable declared as `short` will contain a `signed short int` and a variable declared as `unsigned short` will contain an `unsigned short int`.

3.3.6   24-Bit Integer Data Types

HI-TECH C Compiler for PIC10/12/16 MCUs supports two 24-bit integer types. The type `short long` hold 24-bit two's complement signed integers, representing integral values from -8,388,608 to +8,388,607 inclusive. The type `unsigned short` holds 24-bit unsigned integers, representing integral values from 0 to 16,777,215 inclusive.

All 24-bit integer values are represented in little endian format with the Least Significant Byte at the lower address.

Variables may be declared using the `signed short long int` and `unsigned short long int` keyword sequences, respectively, to hold values of these types. When specifying a `short long int` type, the keyword `int` may be omitted. Thus a variable declared as `short long` will contain a `signed short long int` and a variable declared as `unsigned short long` will contain an `unsigned short long int`.

3.3.7   32-Bit Integer Data Types and Variables

HI-TECH C Compiler for PIC10/12/16 MCUs supports two 32-bit integer types. The type `long` holds a 32-bit two's complement signed integer, representing integral values from -2,147,483,648 to +2,147,483,647 inclusive.The type `unsigned long` holds a 32-bit unsigned integer, representing integral values from 0 to 4,294,967,295 inclusive.

All 32-bit integer values are represented in *little endian* format with the least significant word and Least Significant Byte at the lowest address.

The types `long` and `unsigned long` occupy 32 bits as this is the smallest long integer size allowed by the ANSI standard for C.

Variables may be declared using the `signed long int` and `unsigned long int` keyword sequences, respectively, to hold values of these types. Where only `long int` is used in the declaration, the type will be `signed long`. When specifying this type, the keyword `int` may be omitted. Thus a variable declared as `long` will contain a `signed long int` and a variable declared as `unsigned long` will contain an `unsigned long int`.

3.3.8   Floating Point Types and Variables

Floating point is implemented using either a IEEE 754 32-bit format or a modified (truncated) 24-bit form of this.

The 24-bit format is used for all `float` values. For `double` values, the 24-bit format is the default, or if the `--DOUBLE=24` option is used. The 32-bit format is used for `double` values if the `--DOUBLE=32` option is used.

This format is described in Table 3-3, where:

* Sign is the sign bit which indicates if the number is positive or negative
* The exponent is 8-bits which is stored as excess 127 (i.e. an exponent of 0 is stored as 127).
* Mantissa is the mantissa, which is to the right of the radix point. There is an implied bit to the left of the radix point which is always 1 except for a zero value, where the implied bit is zero. A zero value is indicated by a zero exponent.

The value of this number is $(-1)^{sign}$ x $2^{(exponent-127)}$ x 1**.** mantissa.

TABLE 3-3:   FLOATING-POINT FORMATS

Format	Sign	Biased exponent	Mantissa
IEEE 754 32-bit	x	xxxx xxxx	xxx xxxx xxxx xxxx xxxx xxxx
modified IEEE 754 24-bit	x	xxxx xxxx	xxx xxxx xxxx xxxx

Here are some examples of the IEEE 754 32-bit formats shown in Table 3-4. Note that the Most Significant bit of the mantissa column (i.e. the bit to the left of the radix point) is the implied bit, which is assumed to be 1 unless the exponent is zero (in which case the float is zero).

TABLE 3-4:   FLOATING-POINT FORMAT EXAMPLE IEEE 754

Format	Number	Biased exponent	1.mantissa	Decimal
32-bit	7DA6B69Bh	11111011b	1.0100110101101101 0011011b	2.77000e+37
		(251)	(1.302447676659)	
24-bit	42123Ah	10000100b	1.001001000111010b	36.557
		(132)	(1.142395019531)	

The 32-bit example in Table 3-4 can be calculated manually as follows.

The sign bit is zero; the biased exponent is 251, so the exponent is 251-127=124. Take the binary number to the right of the decimal point in the mantissa. Convert this to decimal and divide it by 2^{23} where 23 is the number of bits taken up by the mantissa, to give 0.302447676659. Add 1 to this fraction. The floating-point number is then given by:

$-1^0 \times 2^{124} \times 1.302447676659$

which becomes:

$1 \times 2.126764793256e+37 \times 1.302447676659$

which is approximately equal to:

2.77000e+37

Variables may be declared using the `float` and `double` keywords, respectively, to hold values of these types. Floating point types are always signed and the `unsigned` keyword is illegal when specifying a floating point type. Types declared as `long double` will use the same format as types declared as `double`.

3.3.9   Structures and Unions

HI-TECH C Compiler for PIC10/12/16 MCUs supports `struct` and `union` types. Structures and unions only differ in the memory offset applied for each member.

These types will be at least 1 byte long. On baseline and mid-range devices, structures and unions must be of a size that allows them to fit within a bank of data memory. Enhanced mid-range devices allow for larger objects to be defined. See **Section 3.3.1 "Object Size"**.

The members of structures and unions may not be objects of type bit, but bit fields are fully supported.

Structures and unions may be passed freely as function arguments and function return values. Pointers to structures and unions are fully supported.

3.3.9.1   BIT-FIELDS IN STRUCTURES

HI-TECH C Compiler for PIC10/12/16 MCUs fully supports bit-fields in structures.

Bit-fields are always allocated within 8-bit words, even though it is usual to use the type `unsigned int` in the definition.

The first bit defined will be the Least Significant bit of the word in which it will be stored. When a bit-field is declared, it is allocated within the current 8-bit unit if it will fit, otherwise a new byte is allocated within the structure. Bit-fields can never cross the boundary between 8-bit allocation units. For example, the declaration:

```
struct {
        unsigned        lo : 1;
        unsigned        dummy : 6;
        unsigned        hi : 1;
} foo;
```

will produce a structure occupying 1 byte. If `foo` was ultimately linked at address 10H, the field `lo` will be bit 0 of address 10H; `hi` will be bit 7 of address 10H. The Least Significant bit of `dummy` will be bit 1 of address 10H and the Most Significant bit of `dummy` will be bit 6 of address 10h.

Unnamed bit-fields may be declared to pad out unused space between active bits in control registers. For example, if `dummy` is never used the structure above could have been declared as:

```
struct {
        unsigned        lo : 1;
        unsigned           : 6;
        unsigned        hi : 1;
} foo;
```

A structure with bit-fields may be initialized by supplying a *comma*-separated list of initial values for each field. For example:

```
struct {
        unsigned        lo  : 1;
        unsigned        mid : 6;
        unsigned        hi  : 1;
} foo = {1, 8, 0};
```

Structures with unnamed bit fields may be initialized. No initial value should be supplied for the unnamed members, for example:

```
struct {
        unsigned        lo  : 1;
        unsigned            : 6;
        unsigned        hi  : 1;
} foo = {1, 0};
```

will initialize the members lo and hi correctly.

3.3.9.2   STRUCTURE AND UNION QUALIFIERS

HI-TECH C supports the use of type qualifiers on structures. When a qualifier is applied to a structure, all of its members will inherit this qualification. In the following example the structure is qualified `const`.

```
const struct {
        int number;
        int *ptr;
} record = { 0x55, &i };
```

In this case, the entire structure will be placed into the program space and each member will be read-only. Remember that all members are usually initialized if a structure is `const` as they cannot be initialized at runtime.

If the members of the structure were individually qualified `const` but the structure was not, then the structure would be positioned into RAM, but each member would be read-only. Compare the following structure with the above.

```
struct {
        const int number;
        int * const ptr;
} record = { 0x55, &i};
```

3.3.10 Standard Type Qualifiers

Type qualifiers provide additional information regarding how an object may be used. HI-TECH C supports both ANSI C qualifiers and additional special qualifiers which are useful for embedded applications and which take advantage of the PIC MCU architecture.

3.3.10.1 CONST TYPE QUALIFIER

HI-TECH C supports the use of the ANSI type qualifiers `const` and `volatile`.

The `const` type qualifier is used to tell the compiler that an object is read only and will not be modified. If any attempt is made to modify an object declared `const`, the compiler will issue a warning or error.

User-defined objects declared `const` are placed in a special psect linked into the program space. Objects qualified `const` may be absolute. The `@ address` construct is used to place the object at the specified address in program memory as in the following example which places the object `tableDef` at address 0x100.

```
const int tableDef[] @ 0x100 = { 0, 1, 2, 3, 4};
```

Usually a `const` object must be initialized when it is declared, as it cannot be assigned a value at any point at runtime. For example:

```
const int  version = 3;
```

will define `version` as being an `int` variable that will be placed in the program memory, will always contain the value 3, and which can never be modified by the program. However uninitialized `const` objects can be defined and are useful if you need to place an object in program memory over the top of other objects at a particular location. Usually uninitialized const objects will be defined as absolute as in the following example.

```
const char checksumRange[0x100] @ 0x800;
```

will define the object checksumRange as a 0x100 byte array of characters located at address 0x800 in program memory. This definition will not place any data in the HEX file.

3.3.10.2 VOLATILE TYPE QUALIFIER

The `volatile` type qualifier is used to tell the compiler that an object cannot be guaranteed to retain its value between successive accesses. This prevents the optimizer from eliminating apparently redundant references to objects declared `volatile` because it may alter the behavior of the program to do so.

Any SFR which can be modified by hardware or which drives hardware is qualified as `volatile`, and any variables which may be modified by interrupt routines should use this qualifier as well. For example:

```
volatile static unsigned int  TACTL @ 0x160;
```

The `volatile` qualifier does not guarantee that any access will be atomic, which is often not the case with the PIC10/12/16 architecture, which can only access a maximum of 1 byte of data per instruction.

The code produced by the compiler to access `volatile` objects may be different to that to access ordinary variables, and typically the code will be longer and slower for `volatile` objects, so only use this qualifier if it is necessary. However failure to use this qualifier when it is required may lead to code failure.

Another use of the `volatile` keyword is to prevent variables being removed if they are not used in the C source. If a non-`volatile` variable is never used, or used in a way that has no effect on the program's function, then it may be removed before code is generated by the compiler.

A C statement that consists only of a `volatile` variable will produce code that reads the variable's memory location and discards the result. For example the entire statement:

```
PORTB;
```

will produce assembly code the reads `PORTB`, but does nothing with this value. This is useful for some peripheral registers that require reading to reset the state of interrupt flags. Normally such a statement is not encoded as it has no effect.

Some variables are treated as being `volatile` even though they may not be qualified in the source code. See **Section 3.9.4.2 "Undefined Symbols"** if you have assembly code in your project.

3.3.11 Special Type Qualifiers

HI-TECH C Compiler for PIC10/12/16 MCUs supports special type qualifiers to allow the user to control placement of `static` and `extern` class variables into particular address spaces.

3.3.11.1 PERSISTENT TYPE QUALIFIER

By default, any C variables that are not explicitly initialized are cleared to zero on startup. This is consistent with the definition of the C language. However, there are occasions where it is desired for some data to be preserved across a reset.

The `persistent` type qualifier is used to qualify variables that should not be cleared by the runtime startup code.

In addition, any `persistent` variables will be stored in a different area of memory to other variables. Different psects are used to hold these objects. See **3.7.1 "Compiler-generated Psects"** for more information.

This type qualifier may not be used on variables of class `auto`, however statically defined local variables may be qualified `persistent`. For example, you should write:

```
void test(void)
{
    static persistent int intvar;  /* must be static */
    // ...
}
```

If the PICC option, `--STRICT` is used, this type qualifier is changed to `__persistent`.

3.3.11.2 NEAR TYPE QUALIFIER

The `near` type qualifier can be used to place static variables in the common memory of the PIC MCU, if such memory is supported by the selected device.

Some of the PIC MCU architectures implement data memory which can be always accessed regardless of the currently selected bank. This *common memory* can be used to reduce code size and execution times as the bank selection instructions that are normally required to access data in banked memory are not required when accessing the common memory. There is very small amounts of this memory, if it is present at all, often only a few bytes.

The compiler automatically uses the common memory for frequently accessed user-defined variables so this qualifier would only be needed for special memory placement of objects, for example if C variables are accessed in hand-written assembly code that assumes that they are located in this memory.

This qualifier is controlled by the compiler option `--ADDRQUAL`, which determines its effect, see **Section 2.7.18 "--ADDRQUAL: Set Compiler Response to Memory Qualifiers"**. Based on this option's settings, this qualifier may be binding or ignored (which is the default operation). Qualifiers which are ignored will not produce an error or warning, but will have no effect.

Here is an example of an `unsigned char` object qualified as `near`:

```
near unsigned char fred;
```

Objects qualified `near` cannot be `auto` or parameters to a function, but can be qualified `static`, allowing them to be defined locally within a function, as in:

```
static near unsigned char local_fred;
```

Note that the compiler may store some temporary objects in the common memory, so not all of this space may be available for user-defined variables.

If the PICC option, `--STRICT` is used, this type qualifier is changed to `__near`.

3.3.11.3   BANK0, BANK1, BANK2 AND BANK3 TYPE QUALIFIERS

The `bank0`, `bank1`, `bank2` and `bank3` type qualifiers are recognized by the compiler and allow some degree of control of the placement of objects in the PIC MCU data memory banks. They can be used to allow portability of legacy code or to define C objects that are assumed to be located in certain memory banks by hand-written assembly code.

These qualifiers are controlled by the compiler option `--ADDRQUAL`, which determines their effect, see **Section 2.7.18 "--ADDRQUAL: Set Compiler Response to Memory Qualifiers"**. Based on this option's settings, these qualifiers may be binding or ignored (which is the default operation). Qualifiers which are ignored will not produce an error or warning, but will have no effect.

Objects qualified with any of these qualifiers cannot be `auto` or parameters to a function, but can be qualified `static`, allowing them to be defined locally within a function, as in:

```
void myFunc(void) {
  static bank1 unsigned char play_mode;
```

If the PICC option, `--STRICT` is used, these qualifiers are changed to `__bank0`, `__bank1`, `__bank2` and `__bank3`.

3.3.11.4   EEPROM TYPE QUALIFIER

The `eeprom` type qualifier is recognized by the compiler and allow some degree of control of the placement of objects in the PIC EEPROM memory, for those devices that implement such memory.

This qualifier is controlled by the compiler option `--ADDRQUAL`, which determines its effect, see **Section 2.7.18 "--ADDRQUAL: Set Compiler Response to Memory Qualifiers"**. Based on this option's settings, this qualifier may be binding or ignored (which is the default operation). Qualifiers which are ignored will not produce an error or warning, but will have no effect.

Objects qualified with this qualifier cannot be `auto` or parameters to a function, but can be qualified `static`, allowing them to be defined locally within a function, as in:

```
void myFunc(void) {
  static eeprom unsigned char inputData[3];
```

If the PICC option, `--STRICT` is used, these qualifiers are changed to `__eeprom`.

3.3.12   Pointer Types

There are two basic pointer types supported by HI-TECH C Compiler for PIC10/12/16 MCUs: data pointers and function pointers. Data pointers hold the address of variables which can be indirectly read, and possible indirectly written, by the program. Function pointers hold the address of an executable function which can be called indirectly via the pointer.

To conserve memory requirements and reduce execution time, pointers on PIC devices are made different sizes and formats. The HI-TECH C Compiler for PIC10/12/16 MCUs compiler uses sophisticated algorithms to track the assignment of addresses to all pointers, and, as a result, no non-standard qualifiers are required when defining pointer variables. Despite this, the size of each pointer is optimal for its intended usage in the program.

3.3.12.1   COMBINING TYPE QUALIFIERS AND POINTERS

It is helpful to first review the ANSI C standard conventions for definitions of pointer types.

Pointers can be qualified like any other C object, but care must be taken when doing so as there are two quantities associated with pointers. The first is the actual pointer itself, which is treated like any ordinary C variable and has memory reserved for it. The second is the target, or targets, that the pointer references, or to which the pointer points. The general form of a pointer definition looks like the following.

```
target_type_&_qualifiers * pointer's_qualifiers pointer's_name;
```

Any qualifiers to the right of the * (i.e. next to the pointer's name) relate to the pointer variable itself. The type and any qualifiers to the left of the * relate to the pointer's targets. This makes sense since it is also the * operator that dereferences a pointer which allows you to get from the pointer variable to its current target.

Here are three examples of pointer definitions using the `volatile` qualifier. The fields in the definitions have been highlighted with spacing:

```
volatile int *          vip ;
int          * volatile ivp ;
volatile int * volatile vivp ;
```

The first example is a pointer called `vip`. It contains the address of `int` objects that are qualified `volatile`. The pointer itself — the variable that holds the address — is *not* `volatile`, however the objects that are accessed when the pointer is dereferenced are treated as being `volatile`. In other words, the target objects accessible via the pointer may be externally modified.

The second example is a pointer called `ivp` which also contains the address of `int` objects. In this example, the pointer itself is `volatile`, that is, the address the pointer contains may be externally modified, however the objects that can be accessed when dereferencing the pointer are not `volatile`.

The last example is of a pointer called `vivp` which is itself qualified `volatile`, and which also holds the address of `volatile` objects.

Bare in mind that one pointer can be assigned the addresses of many objects; for example, a pointer that is a parameter to a function is assigned a new object address every time the function is called. The definition of the pointer must be valid for every target address assigned.

> **Note:**   Care must be taken when describing pointers. Is a "const pointer" a pointer that points to `const` objects, or a pointer that is `const` itself? You can talk about "pointers to const" and "const pointers" to help clarify the definition, but such terms may not be universally understood.

3.3.12.2 DATA POINTERS

HI-TECH C monitors and records *all* assignments of addresses to each data pointer the program contains. This includes assignment of the address of objects to pointers; assignment of one pointer to another; initialization of pointers when they are defined; and takes into account when pointers are ordinary variables and function parameters, and when pointers are used to access basic objects, or structures or arrays.

The size and format of the address held by each pointer is based on this information. When more than one address is assigned to a pointer at different places in the code, a set of all possible targets the pointer can address is maintained. This information is specific to each pointer defined in the program, thus two pointers with the same C type may hold addresses of different sizes and formats due to the way the pointers were used in the program.

The compiler tracks the memory location of all targets, as well as the size of all targets to determine the size and scope of a pointer. The size of a target is important as well particularly with arrays of structures. A pointer must be able to be incremented to point to all the elements of an array, for example.

There are several pointer classifications used with the HI-TECH C Compiler for PIC10/12/16 MCUs, such as those indicated below.

- An 8-bit pointer capable of accessing common memory and two consecutive banks, e.g. banks 0 and 1, or banks 7 and 8, etc
- A 16-bit pointer capable of accessing the entire data memory space;
- An 8-bit pointer capable of accessing up to 256 bytes of program space data;
- A 16-bit pointer capable of accessing up to 64 kbytes of program space data;
- A 16-bit pointer capable of accessing the entire data space memory and up to 64 kbytes of program space data;

Each data pointer will be allocated one of the available classifications after preliminary scans of the source code. There is no mechanism by which the programmer can specify the style of pointer required (other than by the assignments to the pointer). If the C code does not convey the required information to the compiler, then it is not complete or accurate.

Consider the following program in the early stages of development. It consists of the following code.

```
int i, j;

int getValue(const int * ip) {
    return *ip;
}

void main(void) {
    j = getValue(&i);
    // ... code that uses j
}
```

A pointer, `ip`, is a parameter to the function `getValue()`. The pointer target type uses the qualifier `const` since we do not want the pointer to be used to write to any objects whose addresses are passed to the function. The `const` qualification serves no other purpose and does not alter the format of the pointer variable.

If the compiler allocates the variable `i` (defined in `main()`) to bank 0 data memory it will also be noted that the pointer `ip` (parameter to `getValue()`) only points to one object that resides in bank 0 of the data memory. In this case, the pointer, `ip`, is made an 8-bit wide data pointer. The generated code that dereferences `ip` in `getValue()` will be generated assuming that the address can only be to an object in bank 0.

As the program is developed, another variable, x, is defined and (unknown to the programmer) is allocated space in bank 2 data memory. The main() function now looks like:

```
int i, j;  // allocated to bank 0 in this example
int x;     // allocated to bank 2 in this example

int getValue(const int * ip) {
    return *ip;
}

void main(void) {
    j = getValue(&i);
    // ... code that uses j
    j = getValue(&x);
    // ... code that uses j
}
```

The pointer, ip, now has targets that are in bank 0 and in bank 2.To be able to accommodate this situation, the pointer is made 16 bits wide, and the code used to dereference the pointer will change accordingly. This takes place without any modification to the source code.

One positive aspect of tracking pointer targets is less of a dependence on pointer qualifiers. The standard qualifiers const and volatile must still be used in pointer definitions to indicate a read-only or externally-modifiable target object, respectively. However this is in strict accordance with the ANSI C standard. HI-TECH specific qualifiers, like near and the bankx qualifiers, do not need to be used to indicate pointer targets, have no effect, and should be avoided. Omitting these qualifiers will result in more portable and readable code, and lessen the chance of extraneous warnings being issued by the compiler.

3.3.12.3   POINTERS TO BOTH MEMORY SPACES

When a pointer is assigned the address of one or more objects allocated memory in the data space, and also assigned the address of one or more const objects, the pointer will be classified such that it can dereference both memory spaces, and the address will be encoded so that the target memory space can be determined at run-time. This encoding may vary for different target devices.

To extend the example given in **Section 3.3.12.2 "Data Pointers"** the code is now developed further, and the function getValue() is now called with the address of an object that resides in the program memory, as shown.

```
int i, j;  // allocated to bank 0 in this example
int x;     // allocated to bank 2 in this example
const int type = 0x3456;

int getValue(const int * ip) {
    return *ip;
}

void main(void) {
    j = getValue(&i);
    // ... code that uses j
    j = getValue(&x);
    // ... code that uses j
    j = getValue(&type);
    // ... code that uses j
}
```

Again the targets to the pointer, `ip`, are determined and now the pointer is made of the class that can access both data and program memory. The generated code to dereference the pointer will be such that it can determine the required memory space from the address and access either space accordingly. Again, this takes place without any change in the definition of the pointer.

3.3.12.4 SPECIAL POINTER TARGETS

Pointers and integers are not interchangeable. Assigning an integer constant to a pointer will generate a warning to this effect. For example:

```
const char * cp = 0x123;  // the compiler will flag this as bad code
```

There is no information in the integer constant, 0x123, relating to the type, size or memory location of the destination. There is a very good chance of code failure if pointers are assigned integer addresses and dereferenced, particularly for devices like PIC devices which have more than one address space. Is 0x123 an address in data memory or program memory? How big is the object found at address 0x123?

Always take the address of a C object when assigning an address to a pointer. If there is no C object defined at the destination address, then define or declare an object at this address which can be used for this purpose. Make sure the size of the object matches the range of the memory locations that can be accessed.

For example, a checksum for 1000 memory locations starting at address 0x900 in program memory is to be generated. A pointer is used to read this data. You may be tempted to write code such as:

```
const char * cp;
cp = 0x900;  // what resides at 0x900???
```

And increment the pointer over the data. A much better solution is this:

```
const char * cp;
const char inputData[1000] @ 0x900;
cp = &inputData;
// cp is incremented over inputData and used to read values there
```

In this case the compiler can determine the size of the target and the memory space. The array size and type indicates the size of the pointer target, the `const` qualifier on the object (not the pointer) indicates the target is located in program memory space. Note that the `const` array does not need initial values to be specified in this instance, see **Section 3.3.10.1 "Const Type Qualifier"**.

If the pointer has to access objects in data memory you need to define a different object to act as a dummy target. For example, if the checksum was to be calculated over 10 bytes starting at address 0x90 in data memory, the following code could be used.

```
const char * cp;
char inputData[10] @ 0x90;
cp = &inputData;
// cp is incremented over inputData and used to read values there
```

User-defined absolute objects will not be cleared by the runtime startup code and can be placed over the top of other absolute variables.

Take care when comparing (subtracting) pointers. for example:

```
if(cp1 == cp2)
  ; take appropriate action
```

The ANSI C standard only allows pointer comparisons when the two pointer targets are the same object. The address may extend to one element past the end of an array.

Comparisons of pointers to integer constants are even more risky, for example:

```
if(cp1 == 0x246)
    ; take appropriate action
```

In some cases pointers hold an address offset and if the pointer can reference objects in more than one memory space, additional bits in the address will be used to distinguish which memory space is being accessed. Thus a pointer which points to an object stored at address 0x246 in data memory, may contain a different value to a pointer that points to a target located at address 0x246 in program memory. In the above example, the compiler will not know which memory space is being accessed and will not know how to translate the integer into the corresponding format used by `cp1`. Never compare pointers and integer constants.

A NULL pointer is the one instance where a constant value can be assigned to a pointer and this is handled correctly by the compiler. A NULL pointer is numerically equal to 0 (zero), but this is a special case imposed by the ANSI C standard. Comparisons with `NULL` are also allowed.

If `NULL` is the only value assigned to a pointer, the pointer will be made as small as possible.

3.4   STORAGE CLASS AND OBJECT PLACEMENT

Objects are positioned in different memory areas based on their storage class and declaration. This is discussed in the following sections.

3.4.1   Local Variables

A *local variable* is one which only has scope within the block in which it was defined. That is, it may only be referenced within that block. C supports two classes of local variables in functions: `auto` variables which are normally allocated in the function's stack frame, and `static` variables which are always given a fixed memory location and have permanent duration.

3.4.1.1   AUTO VARIABLES

The `auto` (short for *automatic*) variables are the default type of local variable. Unless explicitly declared to be `static`, a local variable will be made `auto`. The `auto` keyword may be used if desired.

These variables are typically stored on a hardware stack, but the PIC10/12/16 devices do not employ a hardware stack for storage of data. A stack is provided for function return addresses, but this can not be used for any other purpose.

On these devices, `auto` variables are allocated memory in a *compiled stack*. This is a fixed block, or blocks, of memory in which the `auto` variables associated with each function are assigned static addresses. Function parameter variables, as well as temporary variables allocated by the compiler, are also placed in this compiled stack. These variables behave in a similar way to `auto` variables.

See **Section 3.4.2 "Compiled Stack Operation"** for detailed information on the compiled stack and how it is formed.

The `auto` variables defined in a function will not necessarily be allocated memory in the order declared, in contrast to parameters which are always allocated memory based on their lexical order. In fact, `auto` variables for one function may be scattered across many RAM banks.

Note that most type qualifiers cannot be used with `auto` variables, since there is no control over the storage location. The exceptions are the standard qualifiers: `const` and `volatile`.

Each `auto` object is referenced in assembly code using a special symbol defined by the code generator. If accessing auto variables defined in C source code, you must use these symbols, which are discussed in **Section 3.9.3.1 "Equivalent Assembly Symbols"**.

3.4.1.2   STATIC VARIABLES

Uninitialized `static` variables are allocated a permanent fixed memory location. Static variables are local in scope to the function in which they are declared, but may be accessed by other functions via pointers since they have permanent duration.

Variables which are `static` are guaranteed to retain their value between calls to a function, unless explicitly modified via a pointer.

Variables which are `static` and which are initialized only have their initial value assigned once during the program's execution. Thus, they may be preferable over initialized `auto` objects which are assigned a value every time the block in they are defined begins execution. Any initialized static variables are initialized in the same way as other non-`auto` initialized objects by the runtime startup code, see **Section 2.4.2 "Runtime Startup Code"**.

Local objects which are `static` are assigned an assembly symbol which consists of the function name followed by an `@` symbol and the variable's lexical name, e.g. `main@foobar` will be the assembly identifier used for the `static` variable `foobar` defined in `main()`.

Non-local static objects use their lexical name with a leading *underscore* character, e.g. `_foobar` will be the assembly identifier used for this object. However, if there is more than one such `static` object defined, then subsequent objects will use the name of the file that contains them and their lexical name separated by an `@` symbol, e.g. `lcd@foobar`. would be the assembly symbol for the `static` variable `foobar` defined in `lcd.c`.

3.4.2   Compiled Stack Operation

Generally speaking, a compiler can either take advantage of a hardware stack that can be implemented using a device's instructions and registers, or produce code which uses a compiled stack to statically allocate `auto` variables.

Baseline and Mid-Range PIC devices can only use their hardware stack for function return addresses and have no instructions to access this stack for variables. The stack size is also only several words long and so it unsuitable for data of any substantial quantity. As a result, HI-TECH C Compiler for PIC10/12/16 MCUs uses a compiled stack for all `auto` variables.

Stored on the compiled stack are the `auto` variables for all functions, plus the parameter variables for each function. A function may also require temporary variables. These are allocated space along with, and are treated in a similar way as, the function's `auto` variables. Local variables which are qualified `static` are not stored on the stack.

A compiled stack consists of fixed memory areas that are usable by each function's stack-based variables. When a compiled stack is used, functions are not re-entrant since stack-based variables in each function will use the same fixed area of memory every time the function is invoked.

Fundamental to the generation of the compiled stack is the call graph, which defines a tree-like hierarchy of function calls, i.e it shows what functions may be called by each function.

There will be one graph produced for each root function. A root function is typically not called, but which is executed via other means and contains a program entry point. The function `main()` is an example of a root function that will be in every project. Interrupt functions which are executed when a hardware interrupt occurs, are another example.

FIGURE 3-1:   FORMATION OF CALL GRAPH
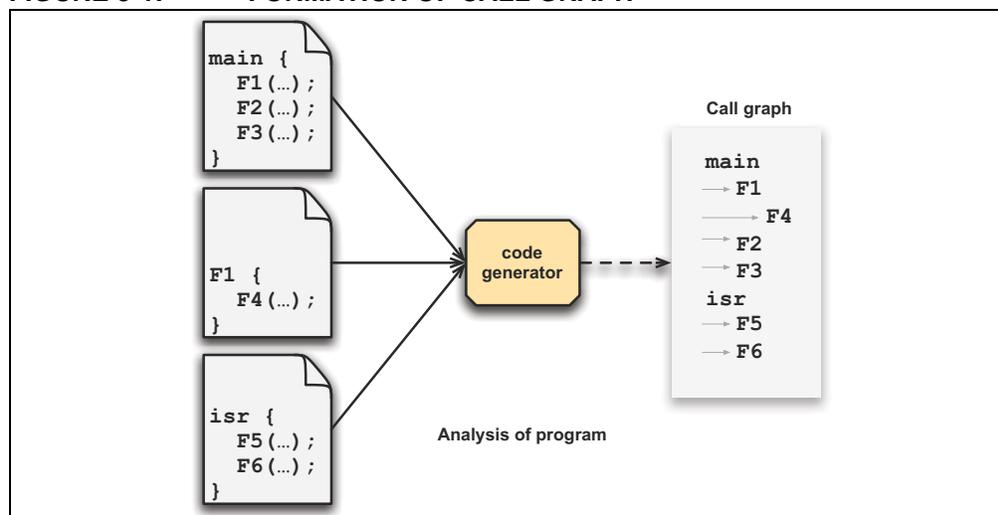
Figure 3-1shows sections of a program being analyzed by the code generator to form a call graph. In the original source code, the function `main()` calls `F1()`, `F2()` and `F3()`. `F1()` calls `F4()`, but the other two functions make no calls. The call graph for `main()` indicates these calls. The symbols `F1`, `F2` and `F3` are all indented one level under main. `F4` is indented one level under `F1`.

This is a static call graph which shows all possible calls. If the exact code for function `F1()` looked like:

```
int F1(void) {
  if(PORTA == 44)
    return F4();
  return 55;
}
```

the function `F4()` will always appear in the call graph, even though it is conditionally executed in the actual source code. Thus, the call graph indicates all functions that *might* be called.

In the diagram, there is also an interrupt function, `isr()`, and it too has a separate graph generated.

The term main-line code is often used, and refers to any code that is executed as a result of the `main()` function being executed. In the above figure, `F1()`, `F2()`, `F3()` and `F4()` are only ever called by main-line code.

The term interrupt code refers to any code that is executed as a result of an interrupt being generated, in the above figure, `F5()` and `F6()` are called by interrupt code.

Figure 3-2 graphically shows an example of how the compiled stack is formed.

FIGURE 3-2: **FORMATION OF THE COMPILED STACK**
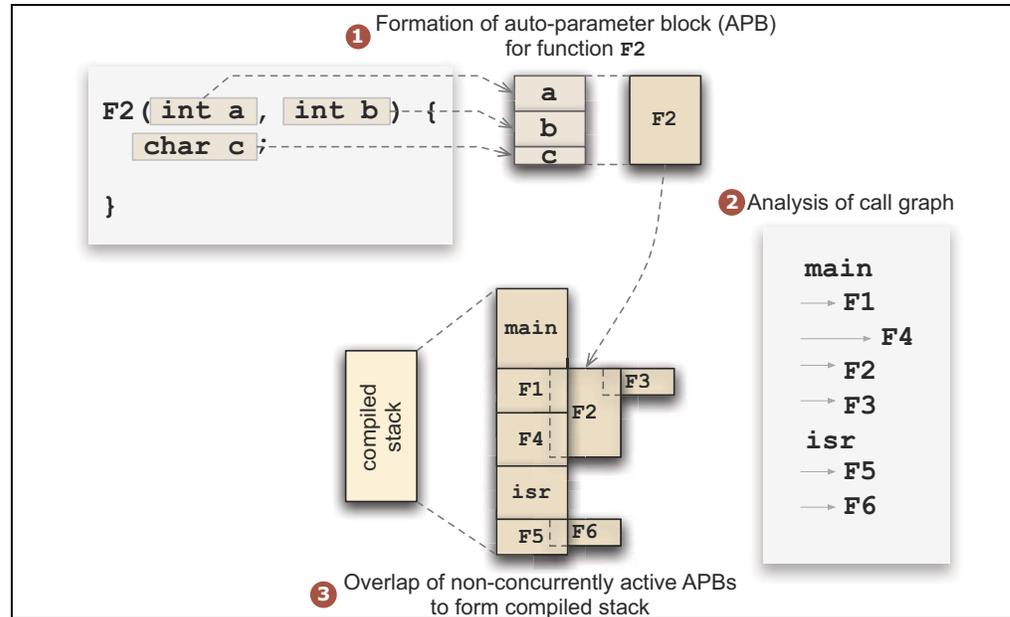
Each function in the program is allocated a block of memory for its parameter, `auto` and temporary variables. Each block is referred to as an auto-parameter block (APB). The figure shows the APB being formed for function `F2()`, which has two parameters, `a` and `b`, and one `auto` variable, `c`.

The parameters to the function are first grouped in an order strictly determined by the lexical order in which they appear in the source code. These are then followed by any `auto` objects, however the `auto` objects may be placed in any order. So we see memory for `a` is followed by that for `b` and lastly `c`.

Once these variables have been grouped, the exact location of each object is not important at this point and we can represent this memory by one block — the APB for this function.

The APBs are formed for all functions in the program. Then, by analyzing the call graph, these blocks are assigned positions, or bases values, in the compiled stack.

Memory can be saved if the following point is observed: If two functions are never active at the same time, then their APBs can be overlapped.

In the example shown in the figure, `F4()` and `F1()` are active at the same time, in fact `F1()` calls `F4()`. However `F2()`, `F3()` and `F1()` are never active at the same time; `F1()` must return before `F2()` or `F3()` can be called by `main()`. The function `main()` will always be active and so its APB can never overlap with that of an other function.

In the compiled stack, you can see that the APB for `main()` is allocated unique memory. The blocks for `F1()`, `F2()` and `F3()` are all placed on top of each other and the same base value in the compiled stack, however the memory taken up by the APBs for `F1()` and `F4()` are unique and do not overlap.

Our example also has an interrupt function, `isr()`, and its call graph is used to assemble the APBs for any interrupt code in the same way. Being the root of a graph, `isr()` will always be allocated unique memory, and the APBs for interrupt functions will be allocated memory following.

The end result is a block of memory which forms the compiled stack. This block can then be placed into the device's memory by the linker.

For devices with more than one bank of data memory, the compiled stack may have components located in more than one bank. The process of building these components of the stack is the same, but each APB will be allocated to one of the stack components based on the remaining memory.

In assembly code, variables within a function's APB are referenced via special symbols, which mark the start of the parameter and temporary area in the APB. An offset is applied to the symbols to locate each variable. Auto variables associated with each function can be placed into a component of the compiled stack in any bank and hence cannot be referenced as an offset to one symbol.

The symbol used to represent the base address of the parameter area within the function's APB is the concatenation of a `?` and the assembler name of the function. The symbol used to represent the base address of the temporary area within the function's APB is the concatenation of `??` and the assembler name of the function.

The parameter variables, a and b, defined in `F2()` in the above figure could be referenced via the assembly symbols: `?_F2` (the Most Significant Byte would be located at `?_F2+1`) and `?_F2+2` (the Most Significant Byte would be located at `?_F2+3`).

Other more human readable symbols are defined by the code generator which can be used in place of these. See **Section 3.9.3.1 "Equivalent Assembly Symbols"** for full information between C domain and assembly domain symbols.

3.4.3 Objects in Program Space

Objects qualified `const` are read-only and are placed in the program memory space. The characters in string literals and also placed in program memory.

On most PIC devices, the program space is not directly readable by the device. The compiler stores data in the program memory by means of `RETLW` instructions which can be called, and which will return a byte if data in the `W` register. The compiler will generate the code necessary to make it appear that program memory is being read directly.

Enhanced Mid-Range PIC devices can directly read their program memory, although the data is also usually stored as `RETLW` instructions. This way the compiler can either produce code that can call these instructions to obtain the program memory data as with the ordinary mid-range devices, or directly read the operand to the instruction (the Least Significant Byte of the `RETLW` instruction). The most efficient access method can be selected by the compiler when the data needs to be read.

A `const` object is usually defined with initial values, as the program cannot write to these objects at runtime. However this is not a requirement. An uninitialized `const` object can be defined to define a symbol, or label, but not make a contribution to the output file. Uninitialized const objects are often made absolute, see **Section 3.4.4 "Absolute Variables"**. Here are examples of `const` object definitions.

```
const char IOtype = 'A';  // initialized const object
const char buffer[10];    // I just define a label
```

See **Section 3.7.1 "Compiler-generated Psects"** for the psects used for these objects.

3.4.4   Absolute Variables

Any non-`auto` variable can be located at an absolute address by following its declaration with the construct `@ address`, where `address` is the location in data memory where the variable is to be positioned. Such a variables is known as an absolute variable. For example:

```
volatile unsigned char Portvar @ 0x06;
```

will declare a variable called `Portvar` located at 06h in the data memory. The compiler will reserve storage for this object and will equate the variable's identifier to that address. The compiler-generated assembler will include a line similar to:

```
_Portvar EQU 06h
```

This construct is primarily intended for equating the address of a C identifier with a special function register, but can be used to place ordinary variables at an absolute address in the general purpose RAM.

The compiler and linker do not make any checks for overlap of absolute variables with other absolute variables so this must be considered when choosing addresses if absolutes are not intended to be overlaid.

> **Note:**   Defining absolute objects can fragment memory and may make it impossible for the linker to position other objects. Avoid absolute objects if at all possible. If absolute objects must be defined, try to place them at either end of a memory bank or page.

When compiling for an enhanced Mid-Range PIC device, the memory allocated for some objects may be spread over multiple RAM banks. Such objects will only ever be accessed indirectly in assembly code, and will use the linear GPR memory implemented on these devices. A linear address (which can be mapped back to the ordinary banked address) will be used with these objects internally by the compiler.

The address specified for absolute objects may either be the traditional banked memory address or the linear address. As the linear addresses start above the largest banked address, it is clear which address is intended. In the following example:

```
int inputBuffer[100] @ 0x2000;
```

it is clear that `inputBuffer` should placed at address 0x2000 in the linear address space, which is address 0x20 in bank 0 RAM in the traditional banked address space. See the device data sheet for exact details regarding your selected device.

Objects qualified `const` can also be made absolute in the same way, however the address will indicate an address in program memory.

Both initialized and uninitialized `const` objects can be made absolute. That latter is useful when you only need to define a label in program memory without making a contribution to the output file.

Variables can also be placed at specific positions by using the `psect` pragma, see **Section 3.10.3.6 "The #pragma psect Directive"**. The decision whether variables should be positioned this way or using absolute variables should be based on the location requirements. Using absolute variables is the easiest method, but only allows placement at an address which must be known prior to compilation. The `psect` pragma is more complex, but offers all the flexibility of the linker to position the new psect into memory. You can, for example, specify that variables reside at a fixed address, or that they be placed after other psects, or that the they be placed anywhere in a compiler-defined or user-defined range of address.

3.5   FUNCTIONS

Functions may be written in the usual way in accordance with the C language. Implementation and special features associated with functions are discussed in the following sections.

3.5.1   Function Size

On baseline PIC devices, the size of the assembly code generated for each function must be less than the size of one page in the target device. A linker "can't find space" error message will be issued if this restriction is not met, and in such cases, the functions must be re-coded or split into several smaller functions.

If the target device is a Mid-Range or enhanced Mid-Range PIC device, this restriction does not apply, and the assembly code generated for each C function may be of any size, limited only by the available program memory.

3.5.2   Absolute Functions

The assembly code associated with a C function can be placed at an absolute address. This can be accomplished by using an `@` *address* construct in a similar fashion to that used with absolute variables. Such functions are called *absolute functions*.

The following example of an absolute function which will place the function at address 400h:

```
int mach_status(int mode) @ 0x400
{
/* function body */
}
```

If you check the assembly list file you will see the function label and the first assembly instruction associated with the function located at 0x400.

If this construct is used with interrupt functions it will only affect the position of the code associated with the interrupt function body. The interrupt context switch code that precedes the function code will not be relocated as it must be linked to the interrupt vector. See also Section **2.7.22 "--CODEOFFSET: Offset Program Code to Address"** for information on how to move reset and interrupt vector locations, which may be useful for designing applications such as bootloaders.

Functions can also be placed at specific positions by using the psect pragma, see **Section 3.10.3.6 "The #pragma psect Directive"**. The decision whether functions should be positioned this way or using absolute functions should be based on the location requirements.

Using absolute functions is the easiest method, but only allows placement at an address which must be known prior to compilation. The `psect` pragma is more complex, but offers all the flexibility of the linker to position the new psect into memory. For example, you can specify that functions reside at a fixed address, or that they be placed after other psects, or that the they be placed anywhere in a compiler-defined or user-defined range of addresses.

3.5.3   Function Argument Passing

The method used to pass function arguments depends on the size and number of arguments involved.

The compiler will either pass parameters in the `W` register, or in the parameter area of the called function.

The parameter area is grouped along with the function's `auto` memory and is placed in the compiled stack. See **Section 3.4.2 "Compiled Stack Operation"** for detailed information on the compiled stack. The parameter variables will be referenced as an offset from the symbol `?_` *function*, where *function* is the name of the function in which the parameter is defined (i.e. the function that is to be called).

If the first parameter is one byte in size, it is passed in the `W` register. All other parameters are passed in the parameter memory. This applies to basic types and to aggregate types like structures.

The parameters for functions that take a variable argument list (defined using an *ellipsis* in the prototype) are placed in the parameter memory, along with named parameters.

Take, for example, the following ANSI-style function.

```
void test(char a, int b);
```

The function `test()` will receive the parameter `b` in its function parameter block and `a` in the `W` register. A call:

```
test(xyz, 8);
```

would generate code similar to:

```
MOVLW     08h        ; move literal 0x8 into...
MOVWF     ?_test     ; the parameter memory
CLRF      ?_test+1    ; locations for the 16-bit parameter
MOVF      _xyz,w     ; move xyz into the W register
CALL      (_test)
```

In this example, the parameter `b` is held in the memory locations `?_test` (Least Significant Byte) and `?_test+1` (Most Significant Byte).

The exact code used to call a function, or the code used to access a parameters from within a function, can always be examined in the assembly list file. See **Section 2.7.17 "--ASMLIST: Generate Assembler List Files"** for the option that generates this file. This is useful if you are writing an assembly routine that must call a function with parameters, or accept arguments when it is called.

3.5.4   Function Return Values

Function return values are passed to the calling function using either the `W` register, or the function's parameter memory. Having return values also located in the parameter memory can reduce the code size for functions that return a modified copy of their parameter.

Eight-bit values are returned from a function in the `W` register.

Values larger than a byte are returned in the function's parameter memory area, with the least significant word in the lowest memory location. For example, the function:

```
int return_16(void)
{
    return 0x1234;
}
```

will exit with the code similar to:

```
MOVLW  34h
MOVWF  (?_return_16)
MOVLW  12h
MOVWF  (?_return_16)+1
RETURN
```

3.5.5 Function Calling

3.5.5.0.1 Baseline Devices

The Baseline PIC devices have a two-level deep hardware stack which is used to store the return addresses of subroutine calls.

Typically, `CALL` instructions are used to transfer control to a C function when it is called, however where the depth of the stack will be exceeded, the compiler will automatically swap to using a method that involves the use of a lookup table and which does not require use of the hardware stack.

When the lookup method is being employed, a function is reached by a jump (not a call) directly to its address. Before this is done the address of a special "return" instruction (implemented as a jump instruction) is stored in a temporary location inside the called function. This return instruction will be able to return control back to the calling function.

This means of calling functions allows functions to be nested deeply without overflowing the stack, however it does come at the expense of memory and program speed.

By default the compiler will determine which functions are permitted to be called via a `CALL` assembly instruction and which will be called via the lookup table. By disabling the `stackcall` suboption to the `--RUNTIME` option all function calls execute via a lookup table unless a function definition is qualified as `fastcall`.

A `fastcall` -qualified function will always be called via a `CALL` instruction. Extreme care must be taken when functions are qualified `fastcall`, since each nested `fastcall` function call will use one word of available stack space. Check the call graph depth in the assembly list file and monitor warning messages from the compiler to ensure that the stack will not overflow. See **Section 3.2.1 "Stack"** for more information on the hardware return address stack.

The function prototype for a Baseline `fastcall` function might look something like:

```
fastcall void my_function(int a);
```

3.5.5.0.2 Mid-Range PIC Devices

The Mid-Range PIC devices have a larger hardware stack and are thus allow a higher degree of function nesting. These devices always use a `CALL` instruction when calling functions and the `fastcall` qualifier has no effect for such functions.

3.5.6 Bank Selection within Functions

A function can return with any RAM bank selected.

The compiler tracks the bank selections made in the generated code associated with each function, even across function calls to other functions. If the bank that is selected when a function returns can be determined, the compiler will use this information to try to remove redundant bank selection instructions which might otherwise be inserted into the generated code.

The compiler will not be able to track the bank selected by routines written in assembly, even if they are called from C code. The compiler will make no assumptions about the selected bank when such routines return.

The "Tracked objects" section associated with each function and which is shown in the assembly list file relates to this bank tracking mechanism.

3.6   OPERATORS

HI-TECH C supports all the ANSI operators. The exact results of some of these are implementation defined. The following sections illustrate code produced by the compiler.

3.6.1   Integral Promotion

When there is more than one operand to an operator, they typically must be of exactly the same type. The compiler will automatically convert the operands, if necessary, so they have the same type. The conversion is to a "larger" type so there is no loss of information.

Even if the operands have the same type, in some situations they are converted to a different type before the operation. This conversion is called *integral promotion*. HI-TECH C performs these integral promotions where required. If you are not aware that the type has changed, the results of some expressions are not what would normally be expected.

Integral promotion is the implicit conversion of enumerated types, `signed` or `unsigned` varieties of `char`, `short int` or bit-field types to either `signed int` or `unsigned int`. If the result of the conversion can be represented by an `signed int`, then that is the destination type, otherwise the conversion is to `unsigned int`.

Consider the following example.

```
unsigned char count, a=0, b=50;
if(a - b < 10)
  count++;
```

The `unsigned char` result of `a - b` is 206 (which is not less than 10), but both `a` and `b` are converted to `signed int` via integral promotion before the subtraction takes place. The result of the subtraction with these data types is -50 (which *is* less than 10) and hence the body of the `if()` statement is executed.

If the result of the subtraction is to be an `unsigned` quantity, then apply a cast. For example:

```
if((unsigned int)(a - b) < 10)
  count++;
```

The comparison is then done using `unsigned int`, in this case, and the body of the `if()` would not be executed.

Another problem that frequently occurs is with the bitwise compliment operator, `~`. This operator toggles each bit within a value. Consider the following code.

```
unsigned char count, c;
c = 0x55;
if( ~c == 0xAA)
  count++;
```

If `c` contains the value 0x55, it often assumed that `~c` will produce 0xAA, however the result is 0xFFAA and so the comparison in the above example would fail. The compiler may be able to issue a mismatched comparison error to this effect in some circumstances. Again, a cast could be used to change this behavior.

The consequence of integral promotion as illustrated above is that operations are not performed with `char` -type operands, but with `int` -type operands. However there are circumstances when the result of an operation is identical regardless of whether the operands are of type `char` or `int`. In these cases, HI-TECH C will not perform the integral promotion so as to increase the code efficiency. Consider the following example.

```
unsigned char a, b, c;
a = b + c;
```

Strictly speaking, this statement requires that the values of b and c should be promoted to unsigned int, the addition performed, the result of the addition cast to the type of a, and then the assignment can take place. Even if the result of the unsigned int addition of the promoted values of b and c was different to the result of the unsigned char addition of these values without promotion, after the unsigned int result was converted back to unsigned char, the final result would be the same. If an 8-bit addition is more efficient than a 16-bit addition, the compiler will encode the former.

If, in the above example, the type of a was unsigned int, then integral promotion would have to be performed to comply with the ANSI C standard.

3.6.2   Shifts applied to integral types

The ANSI standard states that the result of right shifting (>> operator) signed integral types is implementation defined when the operand is negative. Typically, the possible actions that can be taken are that when an object is shifted right by one bit, the bit value shifted into the Most Significant bit of the result can either be zero, or a copy of the Most Significant bit before the shift took place. The latter case amounts to a sign extension of the number.

HI-TECH C Compiler for PIC10/12/16 MCUs performs a sign extension of any signed integral type (for example signed char, signed int or signed long). Thus an object with the signed int value 0x0124 shifted right one bit will yield the value 0x0092 and the value 0x8024 shifted right one bit will yield the value 0xC012.

Right shifts of unsigned integral values always clear the Most Significant bit of the result.

Left shifts (<< operator), signed or unsigned, always clear the Least Significant bit of the result.

3.6.3   Rotation

The C language does not contain a rotate operator. It does allow shifts, as illustrated in **Section 3.6.2 "Shifts applied to integral types"**. However the compiler will detect expressions that implement rotate operations using shift and logical operators.

For the following code:

```
c = (c << 1) | (c >> 7);
```

if c is unsigned and non-volatile, the compiler will detect that the intended operation is a rotate left of 1 bit and will encode the output using the PIC MCU rotate instructions. A rotate left of 2 bits would be implemented with code like:

```
c = (c << 2) | (c >> 6);
```

This code optimization will also work for integral types larger than a char. If the optimization cannot be applied, or this code is ported to another compiler, the rotate will be implemented, but typically with shifts and a bitwise OR operation.

3.6.4 Division and modulus with integral types

The sign of the result of division with integers when either operand is negative is implementation specific. Table 3-5 shows the expected sign of the result of the division of operand 1 with operand 2 when compiled with HI-TECH C.

TABLE 3-5:   INTEGRAL DIVISION

Operand 1	Operand 2	Quotient	Remainder
+	+	+	+
-	+	-	-
+	-	-	+
-	-	+	-

In the case where the second operand is zero (division by zero), the result will always be zero.

3.7   PSECTS

The code generator splits code and data objects into a number of standard "program sections", referred to as *psects*[1]. A psect is just a block of something: a block of code; a block of data etc. By having everything inside a psect, all these blocks can be easily recognized and sorted by the linker, even though they have come from different modules.

One of the main jobs of the linker is to group all the psects from the entire project and place these into the available memory for the device.

A psect can be created in assembly code by using the PSECT assembler directive (see **Section 4.3.9.3 "PSECT"**). The code generator uses this directive to direct assembly code it produces into the appropriate psect.

3.7.1   Compiler-generated Psects

The code generator places code and data into psects with standard names which are subsequent positioned by the default linker options. The linker does not treat these compiler-generated psect any differently to a psect that has been defined by yourself.

Some psects, in particular the data memory psects, use special naming conventions.

For example, take the bss psect. The name bss is historical. It holds uninitialized variables. However there may be some uninitialized variables that will need to be located in bank 0 data memory; others may need to be located in bank 1 memory. As these two groups of variables will need to be placed into different memory banks, they will need to be in separate psects so they can be independently controlled by the linker. In addition, the uninitialized variables that are bit variables need to be treated specially so they need their own psect. So there are a number of different psects that all use the same basename, but which have prefixes and suffixes to make them unique.

The general form of these psect names is:

```
[bit]psectBaseNameCLASS[div]
```

where *psectBaseName* is the base name of the psect, such as bss.The *CLASS* is a name derived from the linker class (see **Section 5.2.1 "-Aclass =low-high,..."**) in which the psect will be linked, e.g. BANK0. The prefix bit is used if the psect holds bit variables. So there may be psects like: bssBANK0, bssBANK1 and bitbssBANK0 defined by the compiler to hold the uninitialized variables.

---

1. Some compilers use the terms *segment*, or *block*, but the concept is the same.

If a psect has to be split into two ranges, then the letters l (elle) and h are used as *div* to indicate if it is the lower or higher division. A psect would be split if memory in the middle of a bank has been reserved, or is in some what not available to position objects. If an absolute variable is defined and is located anywhere inside a memory range, that range will need to be split to ensure that anything in the psects located there do not overwrite the absolute object. Thus you might see bssBANK0l and bssBANK0h psects if a split took place.

The contents of these psects are described below, listed by psect base name.

3.7.1.1   PROGRAM SPACE PSECTS

checksum This is a psect that is used to mark the position of a checksum that has been requested using the --CHECKSUM option, see **Section 2.7.19 "--CHECKSUM: Calculate a checksum"**. The checksum value is added after the linker has executed so you will not see the contents of this psect in the assembly list file, nor specific information in the map file. Linking this psect at a non-default location will have no effect on where the checksum is stored, although the map file will indicate it located at the new address. Do not change the default linker options relating to this psect.

cinit Used by the C initialization runtime startup code. Code in this psect is output by the code generator along with the generated code for the C program and does not appear in the runtime startup assembly module.
This psect can be linked anywhere in the program memory, provided they does not interfere with the requirements of other psects.

config Used to store the configuration words.
This psect must be stored in a special location in the HEX file. Do not change the default linker options relating to this psect.

eeprom Used to store initial values in the EEPROM memory.
Do not change the default linker options relating to this psect.

idata These psects contain the ROM image of any initialized variables. These psects are copied into the data psects at startup. In this case, the class name is used to describe the class of the corresponding RAM-based data psect. These psects will be stored in program memory, not the data memory space. These psects are implicitly linked to a location that is anywhere within the CODE linker class. The linker options can be changed allowing this psect to be placed at any address in the program memory, provided it does not interfere with the requirements of other psects.

idloc Used to store the ID location words.
This psect must be stored in a special location in the HEX file. Do not change the default linker options relating to this psect.

init Used by assembly code in the runtime startup assembly module. The code in this and the cinit define the runtime startup code.
If no interrupt code is defined code from the reset vector may "fall through" into this psect. It is recommended that the default linker options relating to this psect are not changed in case this situation is in effect.

intentry Contains the entry code for the interrupt service routine which is linked to the interrupt vector. This code saves the necessary registers and jumps to the main interrupt code in the case of mid-range devices; for enhanced mid-range devices this psect will contain the interrupt function body.
This psect must be linked at the interrupt vector. Do not change the default linker options relating to this psect. See the --CODEOFFSET option **Section 2.7.22 "--CODEOFFSET: Offset Program Code to Address"** if

you want to move code when using a bootloader.

jmp_tab Only used for the baseline processors, this is a psect used to store jump addresses and function return values.
Do not change the default linker options relating to this psect.

maintext This psect will contain the assembly code for the `main()` function. The code for `main()` is segregated as it contains the program entry point.
Do not change the default linker options relating to this psect as the runtime startup code may "fall through" into this psect which requires that it be linked immediately after this code.

powerup Contains executable code for a user-supplied power-up routine.
Do not change the default linker options relating to this psect.

reset_vec This psect contains code associated with the reset vector.
Do not change the default linker options relating to this psect as it must be linked to the reset vector location of the target device. See the `--CODEOFF-SET` option **Section 2.7.22 "--CODEOFFSET: Offset Program Code to Address"** if you want to move code when using a bootloader.

reset_wrap For baseline PIC devices, this psect contains code which is executed after the device PC has wrapped around to address 0x0 from the oscillator calibration location at the top of program memory.
Do not change the default linker options relating to this psect as it must be linked to the reset vector location of the target device.

strings The `strings` psect is used for `const` objects. It also includes all unnamed string literals. This psect is linked into ROM, since the contents do not need to be modified.
This psect can be linked anywhere in the program memory, provided it does not interfere with the requirements of other psects.

textn These psects (where *n* is a decimal number) contain all other executable code that does not require a special link location.
These psects can be linked anywhere in the program memory, provided they does not interfere with the requirements of other psects.

xxx_text Defines the psect for a function that has been made absolute, i.e. placed at an address. *xxx* will be the assembly symbol associated with the function. For example if the function `rv()` is made absolute, code associated with it will appear in the psect called `_rv_text`.
As these psects are already placed at the address indicated in the C source code, the linker options that position them should not be changed.

3.7.1.2 DATA SPACE PSECTS

nv These psects are used to store persistent variables. They are not cleared or otherwise modified at startup.
These psects may be linked anywhere in their targeted memory bank.

bss These psects contain any uninitialized variables.
These psects may be linked anywhere in their targeted memory bank.

data These psects contain the RAM image of any initialized variables.
These psects may be linked anywhere in their targeted memory bank.

cstack These psects contain the compiled stack. On the stack are auto and parameter variables for the entire program. See **3.4.2 "Compiled Stack Operation"** for information on the compiled stack.

3.8   INTERRUPT HANDLING IN C

The compiler incorporates features allowing interrupts to be fully handled from C code. Interrupt functions are often called *Interrupt Service Routines* (ISRs).

Baseline devices do not utilize interrupts and so the following sections are only applicable for Mid-Range and Enhanced Mid-Range devices.

3.8.1   Interrupt Functions

The function qualifier `interrupt` may be applied to C function definitions to allow them to be called directly from the hardware interrupts. The compiler will process the `interrupt` function differently to any other functions, generating code to save and restore any registers used and return using a special instruction.

If the PICC option `--STRICT` is used, the `interrupt` keyword becomes `__interrupt`.

An interrupt function must be declared as type `void interrupt` and may not have parameters. This is the only function prototype that makes sense for an interrupt function since they are never directly called in the source code.

Interrupt functions must not be called directly from C code (due to the different return instruction that is used), but they themselves may call other functions, both user-defined and library functions.

Mid-Range PIC devices have many sources of interrupt, but only one interrupt vector, and hence should only have one interrupt function defined.

An example of an interrupt function for a Mid-Range PIC MCU processor is shown here.

```
int tick_count;

void interrupt tc_int(void)
{
    if (T0IE && T0IF) {
        T0IF=0;
        ++tick_count;
        return;
    }
    // process other interrupt sources here
}
```

Code will be placed at the interrupt vector which will execute this function after any context switch that is required.

Notice that the code checks for the source of the interrupt, in this case a timer, by looking at the interrupt flag bit (`T0IE`) and the interrupt flag bit (`T0IF`). This is required since interrupt flags associated with a peripheral may be asserted even if the peripheral is not configured to generate an interrupt.

3.8.1.1   CONTEXT SAVING ON INTERRUPTS

Some registers are automatically saved by the hardware when an interrupt occurs. Any registers or compiler temporary objects used by the interrupt function, other than those saved by the hardware, must be saved in software. This is the context save, or context switch code.

Enhanced Mid-Range PIC devices save the `W`, `STATUS`, `BSR` and `FSRx` registers in hardware (using special shadow registers) and hence these registers do not need to be saved by software. In fact, the compiler will never have to produce code to save any registers when compiling for an Enhanced Mid-Range as no additional registers are ever used. This makes interrupt functions on Enhanced Mid-Range PIC devices very fast and efficient.

Other Mid-Range PIC processors only save the entire PC (excluding the `PCLATH` register) on the stack when an interrupt occurs. The the `W`, `STATUS`, `FSR` and `PCLATH` registers and the `BTEMP`[1] pseudo register must be saved by the compiler, if required.

The compiler fully determines which registers and objects are used by an interrupt function, or any of the functions that it calls (based on the known call graph), and saves these appropriately.

Assembly code placed in-line within the interrupt function is not scanned for register usage. Thus, if you include in-line assembly code into an interrupt function, you may have to add extra assembly code to save and restore any registers or locations used. The same is true for any assembly routines called by the interrupt code.

If the `W` register is to be saved by the compiler, it may be stored to memory reserved in the common RAM. If the processor for which the code is written does not have common memory, a byte is reserved in all RAM banks for the storage location for `W` register.

Other registers to be saved are done so in the interrupt function's `auto` area, and thus look like ordinary `auto` variables.

3.8.1.2   CONTEXT RESTORATION

Any objects saved by software are automatically restored by software before the interrupt function returns. The order of restoration is the reverse to that used when context is saved.

3.8.2   Enabling Interrupts

Two macros are available once you have included `<htc.h>` which control the masking of all available interrupts. These macros are `ei()`, which enable or unmask all interrupts, and `di()`, which disable or mask all interrupts.

On all Mid-Range PIC devices, they affect the `GIE` bit in the `INTCON` register. These macros should be used once the appropriate interrupt enable bits for the interrupts that are required in a program have been enabled.

For example:

```
ADIE = 1;  // A/D interrupts will be used
PEIE = 1;  // all peripheral interrupts are enabled
ei();      // enable all interrupts
// ...
di();      // disable all interrupts
```

> **Note:** Never use this macro to re-enable interrupts inside the interrupt function itself. Interrupts are automatically re-enabled by hardware on execution of the `RETFIE` instruction. Re-enabling interrupts inside an interrupt function may result in code failure.

3.8.3   Function Duplication

It is assumed by the compiler that an interrupt may occur at any time. As all functions are not reentrant (because of the dependance on the compiled stack for local objects, see **Section 3.4.2 "Compiled Stack Operation"**), if a function appears to be called by an `interrupt` function and by main-line code this could normally lead to code failure.

HI-TECH C has a feature which will duplicate the output associated with any function called from more than one call tree in the program's call graph. There will be one call tree associated with main-line code, and one tree for the `interrupt` function, if defined.

---

1. The `BTEMP` register is a memory location allocated by the compiler, but which is treated like a register for code generation purposes.

Main-line code will call the original function's output, and the interrupt will call the duplicated function's output. The duplication takes place only in the called function's output; there is no duplication of the C source code itself. The duplicated code and data uses different symbols and are allocated different memory so are fully independent.

This is similar to the process you would need to undertake if this feature was not implemented in the compiler: the C function could be duplicated by hand, given different names and one called from main-line code; the other from the interrupt function. However, you would have to maintain both functions, and the code would need to be reverted if it was ported to a compiler which did support reentrancy.

The duplicate output will have unique identifiers for the assembly symbols used within it. The identifiers consists of the same name used in the original output prefixed with `i1`.

The output of the function called from main-line code will not use any prefixes and the assembly names will be those normally used.

To illustrate, in a program the function `main` calls a function called `input`. This function is also called by an `interrupt` function.

Examination of the assembly list file will show assembly code for both the original and duplicate function outputs. The output corresponding to the C function `input()` will use the assembly label `_input`. The corresponding label used by the duplicate function will be `i1_input`. If the original function makes reference to a temporary variable, the generated output will use the symbol `??_input`, compared to `??i1_input` for the duplicate output. Even local labels within the function output will be duplicated in the same way. The call graph, in the assembly list file, will show the calls made to both of these functions as if they were independently written. These symbols will also be seen in the map file symbol table.

This feature allows the programmer to use the same source code with compilers that use either reentrant or non-reentrant models. It does not handle cases where functions are called recursively.

Code associated with library functions are duplicated in the same way. This also applies to implicitly called library routines, such as those that perform division or floating point operations associated with C operators.

3.8.3.1   DISABLING DUPLICATION

The automatic duplication of the function may be inhibited by the use of a special pragma.

This should only be done if the source code guarantees that an interrupt cannot occur while the function is being called from any main-line code. Typically this would be achieved by disabling interrupts before calling the function. It is not sufficient to disable the interrupts inside the function after it has been called. If an interrupt occurs when executing the function, the code may fail. See **Section 3.8.2 "Enabling Interrupts"** for more information on how interrupts may be disabled.

The pragma is:

```
#pragma interrupt_level 1
```

The pragma should be placed before the definition of the function that is not to be duplicated. The pragma will only affect the first function whose definition follows.

For example, if the function read is only ever called from main-line code when the interrupts are disabled, then duplication of the function can be prevented if it is also called from an interrupt function as follows.

```
#pragma interrupt_level 1
int read(char device)
{
    // ...
}
```

In main-line code, this function would typically be called as follows:

```
di();  // turn off interrupts
read(IN_CH1);
ei();  // re-enable interrupts
```

3.9   MIXING C AND ASSEMBLER CODE

Assembly language code can be mixed with C code using two different techniques: writing assembly code and placing it into a separate assembler module, or including it as in-line assembler in a C module. For the latter, there are two formats in which this can be done, described below.

> **Note:**   The more assembly code a project contains, the more difficult and time con-suming will be its maintenance. As the project is developed, the compiler may work in different ways as some optimizations look at the entire pro-gram. The assembly code is more likely to fail if the compiler is updated due to differences in the way the compiler works. These factors do not affect code written in C.
> If assembly must be added, it is preferable to write this as self-contained routine in a separate assembly module rather than in-lining it in C code.

3.9.1   External Assembly Language Functions

Entire functions may be coded in assembly language as separate .as or .asm source files included into your project. They will be assembled and combined into the output image using the linker. This technique allows arguments and return values to be passed between C and assembler code.

The following are guidelines that must be adhered to when writing a routine in a C-callable assembly routine.

• Select, or define, a suitable psect for the executable assembly code
• Select a name (label) for the routine using a leading underscore character
• Ensure that the routine's label is globally accessible from other modules
• Select an appropriate equivalent C prototype for the routine on which argument passing can be modelled
• Ensure any symbol used to hold arguments to the routine is globally accessible
• Optionally, use a signature value to enable type checking when the function is called
• Write the routine ensuring arguments are read from the correct location, the return value is loaded to the correct storage location before returning, if appropriate
• Ensure any local variables required by the routine have space reserved by the appropriate directive
• Use bank selection instructions and mask addresses of symbols
• If the assembly routine calls other C functions or assembly routines, ensure the compiler is aware of this using the appropriate directives.

The following example goes through these steps. A mapping is performed on the names of all C functions and non-`static` global variables. See **Section 3.9.3.1 "Equivalent Assembly Symbols"** for a complete description of mappings between C and assembly identifiers.

An assembly routine is required which can add two 16-bit values together. The routine must be callable from C code. Both the values are passed in as arguments when the routine is called from the C code. The assembly routine should return the result of the addition as a 16-bit quantity.

Most compiler-generated executable code is placed in psects called `text`*n*, where n is a number. (see **Section 3.7.1 "Compiler-generated Psects"**). We will create our own text psect based on the psect the compiler uses. Check the assembly list file to see how the text psects normally appear. You may see a psect such as the following generated by the code generator.

```
PSECT text0,local,class=CODE,delta=2
```

See **Section 4.3.9.3 "PSECT"** for detailed information on the flags used with the `PSECT` assembler directive. This psect is called `text0`. It is flagged `local`, which means that it is distinct from other psects with the same name. It lives in the `CODE` class. This option is important as it means it will be automatically placed in the area of memory set aside for code. With this flag in place, you do not need to adjust the default linker options to have the psect correctly placed in memory. The last option, the `delta` value, is also very important. This indicates that the memory space in which the psect will be placed is word addressable (value of 2). The PIC10/12/16 program memory space is word addressable; the data space is byte addressable.

We simply need to choose a different name, so we might choose the name `mytext`, as the psect name in which we will place out routine, so we have:

```
PSECT mytext,local,class=CODE,delta=2
```

Let's assume we would like to call this routine `add` in the C domain. In assembly domain we must choose the name `_add` as this then maps to the C identifier `add`. If we had chosen `add` as the assembly routine, then it could never be called from C code. The name of the assembly routine is the label that we will place at the beginning of the assembly code. the label we would use would look like this.

```
_add:
```

We need to be able to call this from other modules, some make this label globally accessible, by using the `GLOBAL` assembler directive (**Section 4.3.9.1 "GLOBAL"**).

```
GLOBAL _add
```

By compiling a dummy C function with a similar prototype to this assembly routine, we can determine the signature value. The C-equivalent prototype to this routine would look like:

```
int add(int, int);
```

Check the assembly list file for the signature value of such a function. Signature values are not mandatory, but allow for additional type checking to be made by the linker. We determine that the following `SIGNAT` directive (**Section 4.3.9.28 "SIGNAT"**) can be used.

```
SIGNAT _add,8298
```

This function will have 4 bytes of parameters, but does not need any local variables (the assembly equivalent of C's `auto` variables). All 4 bytes of parameters will be passed in via memory; the `W` register will not be used for parameter passing. See **Section 3.4.2 "Compiled Stack Operation"** for information on how parameters are passed.

The FNSIZE directive informs the compiler as the memory required for the routine. See **Section 4.3.9.17 "FNSIZE"** for more information on this directive. We would need a directive such as this:

```
FNSIZE _add,0,4
```

to indicate that 4 parameter locations are required, but 0 bytes of autos.

When this routine is called, its parameters will be loaded into the parameter area by the calling function. The return value of the function must be loaded to the same area by our routine before it returns.

Here is an example of the complete routine for a Mid-Range device which could be placed into an assembly file and added to your project. The GLOBAL, FNSIZE and SIGNAT directives do not generator code, and hence do not need to be inside the mytext psect, although you can place them there if you prefer. The BANKSEL directive and BANKMASK macro have been used to ensure that the correct bank was selected and that all addresses are masked to the appropriate size.

```
#include <aspic.h>

GLOBAL _add,?_add  ; make _add globally accessible
SIGNAT _add,8298   ; tell the linker how it should be called
FNSIZE _add,0,4    ; this required 4 bytes of params; no autos

; everything following will be placed into the mytext psect
psect mytext,local,class=CODE,delta=2
; our routine to add to ints and return the result
_add:
    ; params are loaded by the calling function;
    ; we access them here using the special symbol
    BANKSEL   (?_add)              ; select the bank of this object
    MOVF      BANKMASK(?_add+2),w ; take the LSB of the second param
    ADDWF     BANKMASK(?_add),f   ; add to the LSB of the first param
    BTFSC     STATUS,0             ; check for carry
    INCF      BANKMASK(?_add+1),f ; handle carry
    MOVF      BANKMASK(?_add+3),w ; take the MSB of the second param
    ADDWF     BANKMASK(?_add+1),f ; add to the MSB of the first param
    ; the result is already in the required location so we can
    ; just return immediately
    RETURN
```

To compile this, the assembly file must be preprocessed as we have used the C pre-processor #include directive. See **Section 2.7.11 "-P: Preprocess Assembly Files"**.

To call an assembly routine from C code, a declaration for the routine must be provided. This ensures that the compiler knows how to encode the function call in terms of parameters and return values.

Here is a C code snippet that declares the operation of the assembler routine, then calls the routine.

```
// declare the assembly routine so it can be correctly called
extern unsigned int add(unsigned a, unsigned b);

void main(void) {
  int a, result;

  a = read_port();
  result = add(5, a);  // call the assembly routine
}
```

This assembly routine does not call other functions or routines. If it did, we would need to use the FNCALL assembler directive to tell the compiler that this has occurred.

If, for example, this routine called another assembly function called `_wait`, then we would add the directive:

```
FNCALL _add,_wait
```

to our routine. If the call to this other routine is conditional (i.e. it may or may not take place depended on the result of some expression) we would still need to add in this directive.

The directive can be placed anywhere, but we may place it underneath the `FNSIZE` directive.

3.9.2   #asm, #endasm and asm()

PIC MCU instructions may also be directly embedded "in-line" into C code using the directives `#asm`, `#endasm` or the statement `asm();`.

The `#asm` and `#endasm` directives are used to start and end a block of assembly instructions which are to be embedded into the assembly output of the code generator. The `#asm` block is not syntactically part of the C program, and thus it does not obey normal C flow-of-control rules. This means that you should not use this form or in-line assembly inside C constructs like `if()`, `while()` `for()` statements. However this is the easiest means of adding multiple assembly instructions.

The `asm()` statement is used to embed a single assembler instruction. This form looks and behaves like a C statement. Only one assembly instruction may be encapsulated within each `asm() statement`.

You may use the `asm(" ")` form of in-line assembly at any point in the C source code as it will correctly interact with all C flow-of-control structures.

The following example shows both methods used:

```c
unsigned int var;

void main(void)
{
    var = 1;
#asm            // like this...
    BCF 0,3
    BANKSEL(_var)
    RLF (_var)&07fh
    RLF (_var+1)&07fh
#endasm
            // do it again the other way...
    asm(" BCF 0,3" );
    asm("BANKSEL _fvar");
    asm(" RLF (_var)&07fh" );
    asm(" RLF (_var+1)&07fh" );
}
```

When using in-line assembler code, great care must be taken to avoid interacting with compiler-generated code. The code generator cannot scan the assembler code for register usage and so will remain unaware if registers are clobbered or used by the assembly code.

If you are in doubt as to which registers are being used in surrounding code, compile your program with the `--ASMLIST` option (see **Section 2.7.17 "--ASMLIST: Generate Assembler List Files"**) and examine the assembler code generated by the compiler. Remember that as the rest of the program changes, the registers and code strategy used by the compiler will change as well.

3.9.3 Accessing C objects from within Assembly Code

The following sections apply to separate assembly modules, and assembly in-line with C code.

3.9.3.1 EQUIVALENT ASSEMBLY SYMBOLS

Most C symbols map to an corresponding assembly equivalent.

The name of a C function maps to an assembler label that will have the same name, but with an *underscore* prepended. So the function `main()` will define an assembly label `_main`.

This mapping is such that an ordinary symbol defined in the assembly domain cannot interfere with an ordinary symbol in the C domain. So for example, if the symbol `main` is defined in the assembly domain, it is quite distinct to the `main` symbol used in C code and they refer to different locations.

If the C function is qualified `static`, and there is more than one function in the program with exactly the same name, the name of the first function will map to the usual assembly symbol and the subsequent functions will map to a special symbol of the form: *fileName@functionName*, where *fileName* is the name of the file that contains the function, and *functionName* is the name of the function.

For example a program contains the definition for two `static` functions, both called `add`. One lives in the file `main.c` and the other in `lcd.c`. The first function will generate an assembly label `_add`. The second will generate the label `lcd@add`.

The name of a global C variable also maps to an assembler label that will have the same name, but with an *underscore* prepended. So the variable `result` will define an assembly label: `_result`.

If the C variable is qualified `static`, there, again, is a chance that there could be more than one variable in the program with exactly the same C name. The same rules apply to variables as to functions. The name of the first variable will map to a special symbol of the form: *fileName@variableName*, where *fileName* is the name of the file that contains the variable, and *variableName* is the name of the variable. If there is more than one static function with the same name, and they contain static variables of the same name, then the assembly symbol used will be of the form: *fileName@functionName@variableName*.

For example a program contains the definition for two `static` variables, both called `result`. One lives in the file `main.c` and the other in `lcd.c`. The first function will generate an assembly label `_result`. The second will generate the label `lcd@result`.

Parameter and `auto` variables also cannot use a simple mapping as there is quite often more than one auto or parameter variable with the same C name in a program. The parameter area is grouped along with the function's auto memory and is placed in the compiled stack. See **Section 3.4.2 "Compiled Stack Operation"** for detailed information on the compiled stack.

The parameter variables in memory will be referenced in assembler as an offset from the symbol `?_function`, where *function* is the name of the function in which the parameter is defined. Temporary symbols will be referenced in assembler as an offset from the symbol `??_function`. Parameters are always allocated based on the order in which they are defined and so their offset can be easily calculated.

To make accessing of parameter and auto variables easier, special equates are defined which map a unique symbol to the auto-parameter block for each function. The symbol has the form: *functionName@variableName*.

For example the `input()` function defines a parameter variable called `fred`. The compiler may define the following:

```
_input@fred          EQU      ??_input+2
```

In assembly code, the variable fred may either be accessed via the symbol `_input@fred`, or via the expression `??_input+2`. If the code for `input()` changes, the offset applied to the `?_input` symbol may change, so the former symbol is often preferred for in-line assembly code. This `EQU` directive is produced by the code generator, and so will not exist for any hand-written assembly code, but you can easily define your own.

3.9.3.2   ACCESSING REGISTERS FROM ASSEMBLY CODE

If writing separate assembly modules, SFR definitions will not automatically be present. The assembly header file `<aspic.h>` can be used to gain access to these register definitions.

Include the file using the assembler's `INCLUDE` directive, (see **Section 4.3.10.3 "INCLUDE"**) or use the C preprocessor's `#include` directive. If you are using the latter method, make sure you compile with the `-P` driver option to preprocess assembly files, see **Section 2.7.11 "-P: Preprocess Assembly Files"**.

The symbols in this header file look similar to the identifiers used in the C domain when including `<htc.h>`, e.g. `PORTA`, `EECON1` etc. They are different symbols in different domains, but will map to the same memory location.

Bits within registers are defined as the *registerName,bitNumber*. So for example, `RA0` is defined as `PORTA,0`.

Here is an example of an assembly module that uses SFRs.

```
#include <aspic.h>
GLOBAL _setports

PSECT text,class=CODE,local,delta=2
_setports:
    MOVLW      0xAA
    BANKSEL    (PORTA)
    MOVWF      BANKMASK(PORTA)
    BANKSEL    (PORTB)
    BSF        RB1
```

3.9.4   Interaction between Assembly and C Code

HI-TECH C Compiler for PIC10/12/16 MCUs incorporates several features designed to allow C code to obey requirements of user-defined assembly code.

The command-line driver ensures that all user-defined assembly files have been processed first, before compilation of C source files begin. The driver is able to read and analyze certain information in the relocatable object files and pass this information to the code generator. This information is used to ensure the code generator takes into account requirement of the assembly code.

See **Section 2.3.3 "Compilation of Assembly Source"** for further information on the compile sequence.

3.9.4.1   ABSOLUTE PSECTS

Some of the information that is extracted from the relocatable objects relates to absolute psects, specifically psects defined using the `abs` and `ovrld`, `PSECT` flags, see **Section 4.3.9.3 "PSECT"** for information on this directive.

HI-TECH C is able to determine the address bounds of absolute psects and uses this information to ensure that the code generated by the code generator does not use memory required by the assembly code. The code generator will reserve any memory used by the assembly code.

Here is an example of how this works. An assembly code files defines a table that must be located at address 0x110 in the data space. The assembly file contains:

```
PSECT lkuptbl,class=RAM,space=1,abs,ovlrd
ORG 110h
lookup:
    DS 20h
```

An absolute psect always starts at address 0. For such psects, you can specify a non-zero starting address by using the `ORG` directive. See **Section 4.3.9.4 "org"** for important information on this directive.

When the project is compiled, this file is assembled and the resulting relocatable object file scanned for absolute psects. As this psect is flagged as being `abs` and `ovlrd`, the bounds and space of the psect will be noted — in this case a memory range from address 0x110 to 0x12F in memory space 1 is being used. This information is passed to the code generator to ensure that this address range are not used by the C code.

The linker handles all of the allocation into program memory, and so only the psects located in data memory need be defined in this way.

3.9.4.2   UNDEFINED SYMBOLS

If a variable needs to be accessible from both assembly and C source code, it can be defined in assembly code, if required, but it is easier to do so in C source code.

A problem could occur if there is a variable defined in C source, but is only ever referenced in the assembly code. In this case, the code generator would remove the variable believing it is unused. The linker would be unable to resolve the symbol referenced by the assembly code and an error will result.

To work around this issue, HI-TECH C also searches assembly-derived object files for symbols which are undefined. These will be symbols that are used, but not defined, in assembly code. The code generator is informed of these symbols, and if they are encountered in the C code the variable is automatically marked as being `volatile`. This action has the same effect as qualifying the variable `volatile` in the source code, see **Section 3.3.10.2 "Volatile Type Qualifier"**.

Variables qualified as `volatile` will never be removed by the code generator, even if they appear to be unused throughout the program.

For example, if a C program defines a global variable as follows:

```
int input;
```

but this variable is only ever used in assembly code. The assembly module(s) can simply declare this symbol using the `GLOBAL` assembler directive, and then use it.

```
GLOBAL _input, _raster
PSECT text,local,class=CODE,delta=2
_raster:
    MOVF    _input,w
```

The compiler knows of the mapping between the C symbol `input`, and the corresponding assembly symbol `_input` (see **Section 3.9.3.1 "Equivalent Assembly Symbols"**). In this instance the C variable `input` will not be removed and be treated as if it was qualified `volatile`.

3.9.4.3   ASSEMBLY STACK USAGE

Assembly routines can define their own local storage by using the `FNSIZE` directive. Assembly locals are the equivalent of C `auto` and parameter variables.

The code generator keeps track of the size of the compiled stack. To ensure that any contribution to this stack size from assembly code is taken into consideration, the `FNSIZE` directives in assembly code are also scanned and information relating to these passed to the code generator.

The `FNCALL` directives are also processed so that the call graph is accurate. This enables the auto-parameter blocks of assembly routines to be overlapped with those of other functions when possible.

As an example, consider the case when an assembly routine requires 2 local memory locations for storage, but no arguments. The definition for the function looks like:

```
GLOBAL _read
FNSIZE _read,2,0
PSECT text,class=CODE,delta=2
_read:
    MOVF        Portage,w
    MOVWF       ?_read          ;1st byte local storage
    MOVF        PORTB,w
    MOVWF       ?_read+1        ;2nd byte local storage
```

The compiler will take into account the 2 bytes required by this routine when calculating the stack size. If this routine is called from C code, the compiler will automatically see that the routine is part of the call graph when it encodes the call.

If this routine is called from other assembly code, that assembly code must use the `FNCALL` directive to indicate that the call has been made. The code generator cannot scan any assembly code for calls but will know of any calls made in C code. For example, if this routine was called by another assembly routine called `_scan`, then the following would be required somewhere in the code.

```
FNCALL _scan,_read
```

3.10   PREPROCESSING

All C source files are preprocessed before compilation. The preprocessed file is always left behind and will have a `.pre` extension and the same base name as the source file from which it is derived.

The `--PRE` option can be used to preprocess and then stop the compilation. See **Section 2.7.46 "--PRE: Produce Preprocessed Source Code"**.

Assembler files can also be preprocessed if the `-P` driver option is issued. See **Section 2.7.11 "-P: Preprocess Assembly Files"**.

3.10.1 Preprocessor Directives

HI-TECH C accepts several specialized preprocessor directives in addition to the standard directives. All of these are listed in Table 3-6.

Macro expansion using arguments can use the # character to convert an argument to a string, and the ## sequence to concatenate arguments.

TABLE 3-6: PREPROCESSOR DIRECTIVES

Directive	Meaning	Example
#	Preprocessor null directive, do nothing	#
#assert	Generate error if condition false	#assert SIZE > 10
#asm	Signifies the beginning of in-line assembly	#asm MOVLW FFh #endasm
#define	Define preprocessor macro	#define SIZE 5 #define FLAG #define add(a,b) ((a)+(b))
#elif	Short for #else #if	see #ifdef
#else	Conditionally include source lines	see #if
#endasm	Terminate in-line assembly	see #asm
#endif	Terminate conditional source inclusion	see #if
#error	Generate an error message	#error Size too big

#if	Include source lines if constant expression true	#if SIZE < 10
		  c = process(10)
		#else
		  skip();
		#endif

#ifdef	Include source lines if preprocessor symbol defined	#ifdef FLAG
		  do_loop();
		#elif SIZE == 5
		  skip_loop();
		#endif

#ifndef	Include source lines if preprocessor symbol not defined	#ifndef FLAG
		  jump();
		#endif

#include	Include text file into source	#include <stdio.h> #include "project.h"
#line	Specify line number and filename for listing	#line 3 final
#*nn*	(Where *nn* is a number) short for #line *nn*	#20
#pragma	Compiler specific options	Refer to **Section 3.10.3 "Pragma Directives"**
#undef	Undefines preprocessor symbol	#undef FLAG
#warning	Generate a warning message	#warning Length not set

3.10.2 Predefined Macros

The compiler drivers define certain symbols to the preprocessor, allowing conditional compilation based on chip type etc. The symbols listed in Table 3-7 show the more common symbols defined by the drivers.

TABLE 3-7:   PREDEFINED MACROS

Symbol	When set	Usage
HI_TECH_C	Always	To indicate that the compiler in use is HI-TECH C®.
_HTC_VER_MAJOR_	Always	To indicate the integer component of the compiler's version number.
_HTC_VER_MINOR_	Always	To indicate the decimal component of the compiler's version number.
_HTC_VER_PATCH_	Always	To indicate the patch level of the compiler's version number.
_HTC_EDITION_	Always	Indicates which of PRO, Standard or Lite compiler is in use. Values of 2, 1 or 0 are assigned respectively.
__PICC__	Always	Indicates HI-TECH compiler for Microchip PIC10/12/16 in use.
MPC	Always	Indicates compiling for Microchip PIC® MCU family.
_PIC12	If Baseline (12-bit) device	To indicate selected device is a baseline PIC devices.
_PIC14	If Mid-Range (14-bit) device	To indicate selected device is a Mid-Range PIC devices.
_PIC14E	If Enhanced Mid-Range (14 bit) device	To indicate selected device is an Enhanced Mid-Range PIC devices.
COMMON	If common RAM present	To indicate whether device has common RAM area.
BANKBITS	Always	Assigned 0, 1 or 2 to indicate 1, 2 or 4 available banks or RAM.
GPRBITS	Always	Assigned 0, 1 or 2 to indicate 1, 2 or 4 available banks or general purpose RAM.
__MPLAB_ICD__	If compiling for MPLAB® ICD or MPLAB ICD 2 debugger	Assigned 1 to indicate that the code is generated for use with the Microchip MPLAB ICD 1. Assigned 2 for MPLAB ICD 2.
_ROMSIZE	Always	To indicate how many words of program memory are available.
_EEPROMSIZE	Always	To indicate how many bytes of EEPROM are available.
_CHIPNAME	When chip selected	To indicate the specific chip type selected, e.f. _16F877
__FILE__	Always	To indicate this source file being preprocessed.
__LINE__	Always	To indicate this source line number.

TABLE 3-7: PREDEFINED MACROS (CONTINUED)

Symbol	When set	Usage
`__DATE__`	Always	To indicate the current date, e.g. `May 21 2004`
`__TIME__`	Always	To indicate the current time, e.g. `08:06:31.`

Each symbol, if defined, is equated to 1 unless otherwise stated.

3.10.3 Pragma Directives

There are certain compile-time directives that can be used to modify the behavior of the compiler. These are implemented through the use of the ANSI standard `#pragma` facility. The format of a pragma is:

```
#pragma keyword options
```

where *keyword* is one of a set of keywords, some of which are followed by certain *options*. A list of the keywords is given in Table 3-8. Those keywords not discussed elsewhere are detailed below.

TABLE 3-8: PRAGMA DIRECTIVES

Directive	Meaning	Example
`inline`	Specify function is inline	`#pragma inline(fabs)`
`interrupt_level`	Allow call from interrupt and main-line code	`#pragma interrupt_level 1`
`jis`	Enable JIS character handling in strings	`#pragma jis`
`nojis`	Disable JIS character handling (default)	`#pragma nojis`
`pack`	Specify structure packing	`#pragma pack 1`
`printf_check`	Enable printf-style format string checking	`#pragma printf_check(printf) const`
`psect`	Rename compiler-generated psect	`#pragma psect nvBANK0=my_nvram`
`regsused`	Specify registers used by function	`#pragma regsused wreg,fsr`
`switch`	Specify code generation for switch statements	`#pragma switch direct`
`warning`	Control messaging parameters	`#pragma warning disable 299,407`

3.10.3.1 THE #PRAGMA INLINE DIRECTIVE

The `#pragma inline` directive is used to indicate to the compiler that a function will be inlined. The directive is only able to be used on special functions that the code generator will handle specially, e.g the `_delay` function.

> **Note:** Use of this pragma with a user-defined function does *not* mean that function will be in-lined.

3.10.3.2 THE #PRAGMA INTERRUPT_LEVEL DIRECTIVE

The `#pragma interrupt_level` directive can be used to prevent function duplication of functions called from main-line and interrupt code. See **Section 3.8.3.1 "Disabling Duplication"** for more information.

3.10.3.3 THE #PRAGMA JIS AND NOJIS DIRECTIVES

If your code includes strings with two-byte characters in the JIS encoding for Japanese and other national characters, the `#pragma jis` directive will enable proper handling of these characters, specifically not interpreting a *backslash*, \, character when it appears as the second half of a two byte character. The `nojis` directive disables this special handling. JIS character handling is disabled by default.

3.10.3.4 THE #PRAGMA PACK DIRECTIVE

Some MCUs requires word accesses to be aligned on word boundaries. Consequently the compiler will align all word or larger quantities onto a word boundary, including structure members. This can lead to "holes" in structures, where a member has been aligned onto the next word boundary.

This behavior can be altered with this directive. Use of the directive `#pragma pack 1` will prevent any padding or alignment within structures. Use this directive with caution - in general if you must access data that is not aligned on a word boundary you should do so by extracting individual bytes and re-assembling the data. This will result in portable code. Note that this directive must *not* appear before any system header file, as these must be consistent with the libraries supplied.

PIC10/12/16 devices can only perform byte accesses to memory and so do not require any alignment of memory objects. This pragma will have no effect when used.

3.10.3.5 THE #PRAGMA PRINTF_CHECK DIRECTIVE

Certain library functions accept a format string followed by a variable number of arguments in the manner of `printf()`. Although the format string is interpreted at runtime, it can be compile-time checked for consistency with the remaining arguments.

This directive enables this checking for the named function, for example the system header file `<stdio.h>` includes the directive:

```
#pragma printf_check(printf) const
```

to enable this checking for `printf()`. You may also use this for any user-defined function that accepts `printf` -style format strings.

The qualifier following the function name is to allow automatic conversion of pointers in variable argument lists. The above example would cast any pointers to strings in RAM to be pointers of the type (`const char *`)

Note that the warning level must be set to -1 or below for this option to have any visible effect. See **Section 2.7.59 "--WARN: Set Warning Level"**.

3.10.3.6 THE #PRAGMA PSECT DIRECTIVE

Normally the object code generated by the compiler is broken into the standard psects as described in **3.7.1 "Compiler-generated Psects"**. This is fine for most applications, but sometimes it is necessary to redirect variables or code into different psects when a special memory configuration is desired.

Code and data for any of the compiler-generated psects may be redirected using a `#pragma psect` directive. The general form of this pragma looks like:

```
#pragma psect standardPsect=newPsect
```

and instructs the code generator that anything that would normally appear in the standard psect *standardPsect*, will now appear in a new psect called *newPsect*. This psect will be identical to *standardPsect* in terms of its flags and attributes, however will have a unique name. Thus, you can explicitly position this new psect without affecting the placement of anything in the original psect.

If the name of the standard psect that is being redirected contains a counter, e.g. text0, text1, text2 etc, then the placeholder %%u should be used in the name of the psect at the position of the counter, e.g. text%%u. This will match any psect, regardless of the counter value. For example, to remap a C function, you could use:

```
#pragma psect text%%u=lookupfunc
int lookup(char ind)
{
  ...
```

Standard psects that make reference to a bank number are not using a counter and do not need the placeholder to match. For example, the redirect an uninitialized variable from bank 1 memory, use:

```
#pragma psect bssBANK1=sharedObj
int foobar;
```

This pragma affects the entire module in which it is located, regardless of the position of the pragma in the file. Any given psect should only be redirected once in a particular module. That is, you cannot redirect the standard psect for some of the module, then swap back to using the standard psect for the remainder of the source code. The pragma should typically be placed at the top of the source file. It is recommended that the code or variables to be separated be placed in a source file all to themselves so they are easily distinguished.

To determine the psect in which the function or object is normally located, define the function or object in the usual way and without this pragma. Now check the assembly list file (see **4.4 "Assembly List Files"**) to determine in which psect the function or object is normally positioned.

Check either the assembly list file or the map file with the pragma in place to ensure that the mapping has worked as expected and that the function or variable has been linked at the address specified.

Variables can also be placed at specific positions by making them absolute, see **Section 3.4.4 "Absolute Variables"**. The same is also true for functions. See **3.5.2 "Absolute Functions"**. The decision whether functions or variables should be positioned using absolutes or via the psect pragma should be based on the location requirements.

Using absolute functions and variables is the easiest method, but only allows placement at an address which must be known prior to compilation. The psect pragma is more complex, but offers all the flexibility of the linker to position the new psect into memory. For example, you can specify that functions or variables reside at a fixed address, or that they be placed after other psects, or that the psect be placed anywhere in a compiler-defined or user-defined range of address. See **Chapter 5. "Linker"** for the features and options available when linking. See also **2.7.7 "-L-: Adjust Linker Options Directly"** for information on controlling the linker from the driver or in MPLAB IDE.

3.10.3.7   THE #PRAGMA REGSUSED DIRECTIVE

The #pragma regsused directive allows the programmer to indicate register usage for functions that will not be "seen" by the code generator, for example if they were written in assembly code. It has no effect when used with functions defined in C code, but in these cases the register usage of these functions can be accurately determined by the compiler and the pragma is not required.

The compiler will determine only those registers and objects which need to be saved for the particular interrupt function defined and use of this pragma allows the code generator to also determine register usage with routines written in assembly code.

The general form of the pragma is:

```
#pragma regsused routineName registerList
```

where *routineName* is the C equivalent name of the function or routine whose register usage is being defined, and *registerList* is a space-separated list of registers names, as shown in Table 3-9.

Those registers not listed are assumed to be unused by the function or routine. The code generator may use any of these registers to hold values across a function call. Hence, if the routine does in fact use these registers, unreliable program execution may eventuate.

TABLE 3-9:   VALID REGISTER NAMES

Register Name	Description
fsr0, fsr0l, fsr0h	Indirect data pointer
fsr1, fsr1l, fsr1h	Indirect data pointer
wreg	The working register
status	The status register

The register names are not case sensitive and a warning will be produced if the register name is not recognized. A blank list indicates that the specified function or routine uses no registers.

For example, a routine called _search is written in assembly code. In the C source, we may write:

```
extern void search(void);
#pragma regsused search wreg status fsr0
```

to indicate that this routine used the W register, STATUS and FSR0.

3.10.3.8   THE #PRAGMA SWITCH DIRECTIVE

Normally, the compiler chooses how switch statements will be encoded to produce the smallest possible code size. The #pragma switch directive can be used to force the compiler to use one particular method.

The general form of the switch pragma is:

```
#pragma switch switchType
```

where *switch_type* is one of the available switch methods listed in Table 3-10.

TABLE 3-10:   SWITCH TYPES

switch type	description
auto	Use smallest code size method (default)
direct	Table lookup (fixed delay)
simple	Sequential xor method

Specifying the direct option to the #pragma switch directive forces the compiler to generate the table look-up style switch method. The time take to execute each case is the same, so this is useful where timing is an issue, e.g state machines.

This pragma affects all subsequent code.

The auto option may be used to revert to the default behavior.

3.10.3.9   THE #PRAGMA WARNING DIRECTIVE

This pragma allows control over some of the compiler's messages, such as warnings and errors. For full information on the massaging system employed by the compiler, see **Section 2.6 "Compiler Messages"**.

3.10.3.9.1   The Warning Disable Pragma

Some warning messages can be disabled by using the `warning disable` pragma.

This pragma will only affect warnings that are produced by the parser or the code generator, i.e. errors directly associated with C code. The position of the pragma is only significant for the parser, i.e. a parser warning number may be disabled for one section of the code to target specific instances of the warning. Specific instances of a warning produced by the code generator cannot be individually controlled. The pragma will remain in force during compilation of the entire module.

The state of those warnings which have been disabled can preserved and recalled using the `warning push` and `warning pop` pragmas. Pushes and pops can be nested to allow a large degree of control over the message behavior.

The following example shows the warning associated with assigning the address of a `const` object to a pointer to non-`const` objects. Such code normally produces warning number 359.

```
int readp(int * ip) {
    return *ip;
}

const int i = 'd';

void main(void) {
    unsigned char c;
#pragma warning disable 359
    readp(&i);
#pragma warning enable 359
}
```

This same affect would be observed using the following code.

```
#pragma warning push
#pragma warning disable 359
    readp(&i);
#pragma warning pop
```

Here the state of the messaging system is saved by the `warning push` pragma. Warning 359 is disabled, then after the source code which triggers the warning, the state of the messaging system is retrieved by using the `warning pop` pragma.

3.10.3.9.2   The Warning Error/warning Pragma

It is also possible to change the type of some messages.

This is only possible by the use of the `warning pragma` and only affects messages generated by the parser or code generator. The position of the pragma is only significant for the parser, i.e. a parser message number may have its type changed for one section of the code to target specific instances of the message. Specific instances of a message produced by the code generator cannot be individually controlled. The pragma will remain in force during compilation of the entire module.

The following shows the warning produced in the previous example being converted to an error for the instance in the function `main()`.

```
void main(void) {
    unsigned char c;
#pragma warning error 359
    readp(&i);
}
```

Compilation of this code would result in an error, not the usual warning. The error will force compilation to cease after the current module has concluded, or immediately if the maximum error count has been reached.

3.11   LINKING PROGRAMS

The compiler will automatically invoke the linker unless the compiler has been requested to stop after producing an intermediate file.

The linker will run with options that are obtained from the command-line driver. These options specify the memory of the device and how the psects should be placed in the memory. No linker scripts are used.

The linker options passed to the linker can be adjusted by the user, but this is only required in special circumstances. See **Section 2.7.7 "-L-: Adjust Linker Options Directly"** for more information.)

The linker creates a map file which details the memory assigned to psects and some objects within the code. The map file is the best place to look for memory information. See **Section 5.4 "Map Files"** for a detailed explanation of the detailed information in this file.

3.11.1   Replacing Library Modules

HI-TECH C comes with a librarian, LIBR, which allows you to unpack a library file and replace modules with your own modified versions. See **Section 6.2 "Librarian"**. However, you can easily replace a library module that is linked into your program without having to do this.

If you add to your project a source file which contains the definition for a routine with the same name as a library routine, then the library routine will be replaced by your routine. This works due to the way the compiler scans source and library files.

When trying to resolve a symbol (a function name, or variable name, for example) the compiler first scans all the source modules for the definition. Only if it cannot resolve the symbol in these files does it then search the library files.

If the symbol is defined in both a source file and a library file, the compiler will never actually search the libraries for this symbol and no error will result. This may not be true if a symbol is defined twice in source files and an error may result if there is a conflict in the definitions.

All libraries are written C code, and the p-code libraries that contain these library routines are actually passed to the code generator, not the linker, but both these applications work in the way described above in resolving library symbols.

You cannot replace a C library function with an equivalent written in assembly code using the above method. If this is required, you will need to use the librarian to edit or create a new library file.

3.11.2   Signature Checking

The compiler automatically produces signatures for all functions. A signature is a 16-bit value computed from a combination of the function's return data type, the number of its parameters and other information affecting the calling sequence for the function. This signature is output in the object code of any function referencing or defining the function.

At link time the linker will report any mismatch of signatures. HI-TECH C is only likely to issue a mismatch error from the linker when the routine is either a precompiled object file or an assembly routine. Other function mismatches are reported by the code generator.

It is sometimes necessary to write assembly language routines which are called from C using an extern declaration. Such assembly language functions should include a signature which is compatible with the C prototype used to call them. The simplest

method of determining the correct signature for a function is to write a dummy C function with the same prototype and check the assembly list file using the `--ASMLIST` option (see **Section 2.7.17 "--ASMLIST: Generate Assembler List Files"**).

For example, suppose you have an assembly language routine called `_widget` which takes two `int` arguments and returns a `char` value. The prototype used to call this function from C would be:

```
extern char widget(int, int);
```

Where a call to `_widget` is made in the C code, the signature for a function with two `int` arguments and a `char` return value would be generated. In order to match the correct signature, the source code for `widget` needs to contain an assembler `SIGNAT` directive which defines the same signature value. To determine the correct value, you would write the following code:

```
char widget(int arg1, int arg2)
{
}
```

The resultant assembler code seen in the assembly list file includes the following line:

```
SIGNAT  _widget,8249
```

The `SIGNAT` directive tells the assembler to include a record in the `.obj` file which associates the value 8249 with symbol `_widget`. The value 8249 is the correct signature for a function with two `int` arguments and a `char` return value.

If this directive is copied into the assembly source file which contains the `_widget` code, it will associate the correct signature with the function and the linker will be able to check for correct argument passing.

If a C source file contains the declaration:

```
extern char widget(long);
```

then a different signature will be generated and the linker will report a signature mis-match which will alert you to the possible existence of incompatible calling conventions.

3.11.3   Linker-Defined Symbols

The linker defines some special symbols that can be used to determine where some psects where linked in memory. These symbols can be used in code, if required.

The link address of a psect can be obtained from the value of a global symbol with name `__L`*name* where *name* is the name of the psect. For example, `__LbssBANK0` is the low bound of the `bssBANK0` psect.

The highest address of a psect (i.e. the link address plus the size) is represented by the symbol `__H`*name*.

If the psect has different load and link addresses, the load start address is represented by the symbol `__B`*name*.

Not all psects are assigned these symbols, in particular those that are not placed in memory by a `-P` linker option (not driver option). See **Section 5.2.18 "-Pspec"**. Psect names may change from one device to another.

Assembly code can use these symbol by globally declaring them, for example:

```
GLOBAL __Lidata
```

NOTES:

Chapter 4. Macro Assembler

The macro assembler included with HI-TECH C PRO for PIC10/12/16 MCU Family assembles source files for PIC 10/12/14/16/17 MCUs. This chapter describes the usage of the assembler and the directives (assembler pseudo-ops and controls) accepted by the assembler in the source files.

Although the term "assembler" is almost universally used to describe the tool which converts human-readable mnemonics into machine code, both "assembler" and "assembly" are used to describe the source code which such a tool reads. The latter is more common and is used in this manual to describe the language. Thus you will see the terms *assembly language* (or just *assembly*), *assembly listing* and etc, but *assembler options*, *assembler directive* and *assembler optimizer*.

4.1 ASSEMBLER USAGE

The assembler is called ASPIC and is available to run on *Windows*, *Linux* and *Mac OS X* systems. Note that the assembler will not produce any messages unless there are errors or warnings — there are no "assembly completed" messages.

Typically the command-line driver, PICC, is used to invoke the assembler as it can be passed assembler source files as input, however the options for the assembler are supplied here for instances where the assembler is being called directly, or when they are specified using the command-line driver option --SETOPTION, see
Section 2.7.53 "--SETOPTION: Set The Command-line Options for Application".

The usage of the assembler is similar under all of available operating systems. All command-line options are recognized in either upper or lower case. The basic command format is shown:

```
ASPIC [ options  ] file
```

files is a space-separated list of one or more assembler source files. Where more than one source file is specified, the assembler treats them as a single module, i.e. a single assembly will be performed on the concatenation of all the source files specified. The files must be specified in full, no default extensions or suffixes are assumed.

options is an optional space-separated list of assembler options, each with a *minus sign* – as the first character in the case of single letter options, or two *minus signs* in the case of multi-letter options. The assembler options must be specified on the command line before any files.

A full list of possible options is given in Table 4-1, and a full description of each option follows.

TABLE 4-1: ASPIC COMMAND-LINE OPTIONS

Option	Meaning	Default
-A	Produce assembler output	Produce object code
-C	Produce cross-reference file	No cross reference
-C*chipinfo*	Define the chipinfo file	dat\picc.ini
-E[*file* | *digit*]	Set error destination/format	
-F*length*	Specify listing page length	66
-H	Output HEX values for constants	Decimal values
-I	List macro expansions	Don't list macros
-L[*listfile*]	Produce listing	No listing
-N	Disable merging optimizations	Merging optimizations enabled
-O	Perform optimization	No optimization
-O*outfile*	Specify object name	srcfile.obj
-R	Specify non-standard ROM	
-T*width*	Specify listing page width	80
-V	Produce line number info	No line numbers
-VER=*version*	Specify full version information for list file title	
-W*level*	Set warning level threshold	0
-X	No local symbols in OBJ file	
--CHIP=*device*	Specify device name	
--DISL=*list*	Specify disabled messages	No message disabled
--EDF=*path*	Specify message file location	
--EMAX=*number*	Specify maximum number of errors	10
--OPT=*optimization*	Specify optimization type	
-VER	Print version number and stop	

4.2 OPTIONS

The command line options recognized by ASPIC are described in the followings sections.

4.2.1   -A: Generate Assembly File

An assembler file will be produced if this option is used rather than the usual object file format. This is useful when checking the optimized assembler produced using the -O optimization option.

By default the output file will an extension .opt, unless the -O*outfile* output option is used to specify another name.

4.2.2   -C: Produce Cross Reference File

A cross reference file will be produced when this option is used. The cross reference file, called *srcfile*.crf, where *srcfile* is the base portion of the first source file name, will contain raw cross reference information. The cross reference utility CREF must then be run to produce the formatted cross reference listing. See **Section 6.4 "Cref"** for more information on this application.

4.2.3   -C: Specify Chip Info File

Specify the chipinfo file to use. The chipinfo file is called `picc.ini` and can be found in the `dat` directory in the compiler's installation directory. This file specifies information about the currently selected device.

4.2.4   -E: Specify Error Format/File

The default format for an error message is in the form:

filename: *line*: *message*

where the error of type *message* occurred on line *line* of the file *filename*. The

`-E` option with no argument will make the assembler use an alternate format for error and warning messages.

Specifying a filename as argument will force the assembler to direct error and warning messages to a file with the name specified.

4.2.5   -F: Specify Page Length

By default the assembly listing format is pageless, i.e. the assembler listing output is continuous. The output may be formatted into pages of varying lengths. Each page will begin with a header and title, if specified.

The `-F` option allows a page length to be specified. A zero value of *length* implies pageless output. The length is specified in a number of lines.

4.2.6   -H: Print Hexadecimal Constant

This option specifies that output constants should be shown as hexadecimal values rather than decimal values. This option affects both the assembly list file, as well as assembly output, when requested.

4.2.7   -I: List Macro Expansions

This option forces listing of macro expansions and unassembled conditionals which would otherwise be suppressed by a `NOLIST` assembler control, see **Section 4.3.10 "Assembler Controls"**. The `-L` option is still necessary to produce an actual listing output.

4.2.8   -L: Generate an Assembly Listing

This option requests the generation of an assembly listing file. If *listfile* is specified then the listing will be written to that file, otherwise it will be written to the standard output.

This option is applied if compiling using PICC, the command-line driver and the `--ASMLIST` driver option, see **Section 2.7.17 "--ASMLIST: Generate Assembler List Files"**.

4.2.9   -O: Optimize assembly

This requests the assembler to perform optimization on the assembly code. Note that the use of this option slows the assembly process down, as the assembler must make an additional pass over the input code.

Debug information for assembler code generated from C source code may become unreliable in debuggers when this option is used.

This option can be applied if compiling using PICC, the command-line driver and the `--OPT` driver option, see **Section 2.7.42 "--OPT: Invoke Compiler Optimizations"**.

4.2.10   -O: Specify Output File

By default the assembler determines the name of the object file to be created by stripping any suffix or extension from the first source filename and appending `.obj`. The `-O` option allows the user to override the default filename and specify a new name for the object file.

4.2.11   -T: Specify Listing Page Width

This option allows specification of the assembly list file page width, in characters. `width` should be a decimal number greater than 41. The default width is 80 characters.

4.2.12   -V: Produce Assembly Debug Information

This option will include line number and filename information in the object file produced by the assembler. Such information may be used by debuggers.

Note that the line numbers will correspond with assembler code lines in the assembler file. This option should not be used when assembling an assembly file produced by the code generator. In that case, debug information should relate back to the original C source, not the intermediate assembly code.

4.2.13   -VER:Specify Version Information

This option allows the full version information, including optional text to indicate beta builds or release candidate builds, to be passed to the assembler. This information is only used in the title of the assembly list file and is not reflected in the output to the `--VER` option.

4.2.14   -X: Strip Local Symbols

The object file created by the assembler contains symbol information, including local symbols, i.e. symbols that are neither public or external. The `-X` option will prevent the local symbols from being included in the object file, thereby reducing the file size. description

4.2.15   --CHIP: Specify Device Name

This option defines the processor which is being used. The processor type can also be indicated by use of the `PROCESSOR` directive in the assembler source file, see **Section 4.3.9.27 "PROCESSOR"**. You can also add your own processors to the compiler via the compiler's chipinfo file.

This option is applied if compiling using PICC, the command-line driver and the `--CHIP` driver option, see **Section 2.7.20 "--CHIP: Define Processor"**.

4.2.16   --DISL: Disable Messages

This option is mainly used by the command-line driver, PICC, to disable particular message numbers. It takes a *comma*-separate list of message numbers that will be disabled during compilation.

This option is applied if compiling using PICC, the command-line driver and the `--MSGDISABLE` driver option, see **Section 2.7.37 "--MSGDISABLE: Disable Warning Messages"**.

See **Section 2.6 "Compiler Messages"** for full information about the compiler's messaging system.

4.2.17   --EDF: Set Message File Path

This option is mainly used by the command-line driver, PICC, to specify the path of the message description file. The default file is located in the `dat` directory in the compiler's installation directory.

See **Section 2.6 "Compiler Messages"** for full information about the compiler's messaging system.

4.2.18   --EMAX: Specify Maximum Number of Errors

This option is mainly used by the command-line driver, PICC, to specify the maximum number of errors that can be encountered before the assembler terminates. The default number is 10 errors.

This option is applied if compiling using PICC, the command-line driver and the `--ERRORS` driver option, see **Section 2.7.28 "--ERRORS: Maximum Number of Errors"**.

See **Section 2.6 "Compiler Messages"** for full information about the compiler's messaging system.

4.2.19   --OPT: Specify Optimization Type

This option complements the assembler `-O` option and indicates specific information about optimizations required. The suboptions: `speed`, `space` and `debug` may be specified to indicate preferences related to procedural abstraction.

Abstraction is enabled when the `space` option is set; disabled when `speed` is set. The `debug` suboption limits the application of some optimizations which otherwise may severely corrupt debug information used by debuggers.

4.2.20   --VER: Print Version Number

This option printed information relating to the version and build of the assembler. The assembler will terminate after processing this option, even if other options and files are present on the command line.

4.3   HI-TECH C ASSEMBLY LANGUAGE

The source language accepted by the macro assembler, ASPIC, is described below. All opcode mnemonics and operand syntax are specific to the PIC10, PIC12, PIC14000, PIC16 and PIC17 devices, include the enhanced mid-range devices. Although the PIC17 family instruction set is supported at the assembler level, the code generator cannot produce code for these devices so no C projects can target these devices.

Additional mnemonics and assembler directives are documented in this section.

4.3.1   Assembler Format Deviations

The HI-TECH PICC assembler uses a slightly modified form of assembly language to that specified by the Microchip data sheets.

4.3.1.1 DESTINATION LOCATION

Certain PIC instructions use the operands ", 0" or ", 1" to specify the destination for the result of that operation. ASPIC uses the more-readable operands ", w" and ", f" to specify the destination register.

The W register is selected as the destination when using the ", w" operand, and the file register is selected when using the ", f" operand or if no destination operand is specified. The case of the letter in the destination operand in not important.

The numerical destination operands cannot be used with ASPIC.

4.3.1.2 LONG JUMPS AND CALLS

The assembler also recognizes several mnemonics which expand into regular PIC MCU assembly instructions. The mnemonics are FCALL and LJMP. These instructions expand into regular CALL and GOTO instructions respectively, but also include the instructions necessary to set the bits in PCLATH (for mid-range devices), or STATUS (for baseline devices) when the destination is in another page of program memory.

These additional mnemonics should be used where possible as they make assembler code independent of the final position of the routines that are to be executed. If the call or jump is determined to be within the current page, the additional code to set the PCLATH bits may be optimized away.

The following example shows an FCALL instruction in the assembly list file. You can see that the FCALL instruction has expanded to five instructions. In this example there are two bit instructions which set/clear bits in the PCLATH register. Bits are also set/cleared in this register after the call to reselect the page which was selected before the fcall.

```
13  0079  3021                         movlw   33
14  007A  120A  158A  2000             fcall   _phantom
          120A  118A
15  007F  3400                         retlw   0
```

4.3.2 Statement Formats

Legal statement formats are shown in Table **Section Table 4-2: "ASPIC statement formats"**.

The *label* field is optional and, if present, should contain one identifier. A label may appear on a line of its own, or precede a mnemonic as shown in the second format.

The third format is only legal with certain assembler directives, such as MACRO, SET and EQU. The *name* field is mandatory and should also contain one identifier.

If the assembly file is first processed by the C preprocessor, see **Section 2.7.11 "-P: Preprocess Assembly Files"**, then it may also contain lines that form valid preprocessor directives. See **Section 3.10.1 "Preprocessor Directives"** for more information on the format for these directives.

There is no limitation on what column or part of the line in which any part of the statement should appear.

TABLE 4-2: ASPIC STATEMENT FORMATS

Format #	Feild1	Field2	Field3	Field4
Format 1	*label:*			
Format 2	*label:*	*mnemonic*	*operands*	*; comment*
Format 3	*name*	*pseudo-op*	*operands*	*; comment*
Format 4	*; comment only*			
Format 5	*empty line*			

4.3.3   Characters

The character set used is standard 7 bit ASCII. Alphabetic case is significant for identifiers, but not mnemonics and reserved words. Tabs are treated as equivalent to spaces.

4.3.3.1   DELIMITERS

All numbers and identifiers must be delimited by white space, non-alphanumeric characters or the end of a line.

4.3.3.2   SPECIAL CHARACTERS

There are a few characters that are special in certain contexts. Within a macro body, the character `&` is used for token concatenation. To use the bitwise `&` operator within a macro body, escape it by using `&&` instead. In a macro argument list, the angle brackets `<` and `>` are used to quote macro arguments.

4.3.4   Comments

An assembly comment is initiated with a semicolon that is not part of a string or character constant.

If the assembly file is first processed by the C preprocessor, see **Section 2.7.11 "-P: Preprocess Assembly Files"**, then the file may also contain C or C++ style comments using the standard `/* ... */` and `//` syntax.

4.3.4.1   SPECIAL COMMENT STRINGS

Several comment strings are appended to assembler instructions by the code generator. These are typically used by the assembler optimizer.

The comment string `;volatile` is used to indicate that the memory location being accessed in the commented instruction is associated with a variable that was declared as `volatile` in the C source code. Accesses to this location which appear to be redundant will not be removed by the assembler optimizer if this string is present.

This comment string may also be used in assembler source to achieve the same effect for locations defined and accessed in assembly code.

4.3.5   Constants

4.3.5.1   NUMERIC CONSTANTS

The assembler performs all arithmetic with signed 32-bit precision.

The default radix for all numbers is 10. Other radices may be specified by a trailing base specifier as given in Table 4-3.

TABLE 4-3:   ASPIC NUMBERS AND BASES

Radix	Format
Binary	Digits 0 and 1 followed by `B`
Octal	Digits 0 to 7 followed by `O`, `Q`, `o` or `q`
Decimal	Digits 0 to 9 followed by `D`, `d` or nothing
Hexadecimal	Digits 0 to 9, A to F preceded by `Ox` or followed by `H` *or* `h`

Hexadecimal numbers must have a leading digit (e.g. 0ffffh) to differentiate them from identifiers. Hexadecimal digits are accepted in either upper or lower case.

Note that a binary constant must have an upper case `B` following it, as a lower case `b` is used for temporary (numeric) label backward references.

In expressions, real numbers are accepted in the usual format, and are interpreted as IEEE 32-bit format.

4.3.5.2   CHARACTER CONSTANTS AND STRINGS

A character constant is a single character enclosed in single quotes '.

Multi-character constants, or strings, are a sequence of characters, not including carriage return or newline characters, enclosed within matching quotes. Either single quotes ' or double quotes " maybe used, but the opening and closing quotes must be the same.

4.3.6   Identifiers

Assembly identifiers are user-defined symbols representing memory locations or numbers. A symbol may contain any number of characters drawn from the alphabetics, numerics and the special characters dollar, $, question mark, ? and underscore, _.

The first character of an identifier may not be numeric. The case of alphabetics is significant, e.g. `Fred` is not the same symbol as `fred`. Some examples of identifiers are shown here:

```
An_identifier
an_identifier
an_identifier1
$
?$_12345
```

4.3.6.1   SIGNIFICANCE OF IDENTIFIERS

Users of other assemblers that attempt to implement forms of data typing for identifiers should note that this assembler attaches no significance to any symbol, and places no restrictions or expectations on the usage of a symbol.

The names of psects (program sections) and ordinary symbols occupy separate, overlapping name spaces, but other than this, the assembler does not care whether a symbol is used to represent bytes, words or sports cars. No special syntax is needed or provided to define the addresses of bits or any other data type, nor will the assembler issue any warnings if a symbol is used in more than one context. The instruction and addressing mode syntax provide all the information necessary for the assembler to generate correct code.

4.3.6.2   ASSEMBLER-GENERATED IDENTIFIERS

Where a `LOCAL` directive is used in a macro block, the assembler will generate a unique symbol to replace each specified identifier in each expansion of that macro. These unique symbols will have the form ??*nnnn* where *nnnn* is a 4 digit number. The user should avoid defining symbols with the same form.

4.3.6.3   LOCATION COUNTER

The current location within the active program section is accessible via the symbol $. This symbol expands to the address of the currently executing instruction (which is different to the address contained in the program counter when executing this instruction). Thus:

```
GOTO $
```

will represent code that will jump to itself and form an endless loop. By using this symbol and an offset, a relative jump destination can be specified.

The address represented by $ is a word address (baseline and Mid-Range devices use program memory which is word-addressable) and thus any offset to this symbol represents a number of instructions. For example:

```
GOTO        $+2
MOVLW       8
MOVWF       _foo
```

will skip one instruction.

4.3.6.4   REGISTER SYMBOLS

Code in assembly modules may gain access to the special function registers by including pre-defined assembly header files. The appropriate file can be included by add the line:

```
#include <aspic.h>
```

to the assembler source file. Note that the file must be included using a C pre-processor directive and hence the option to pre-process assembly files must be enabled when compiling, see **Section 2.7.11 "-P: Preprocess Assembly Files"**. This header file contains appropriate commands to ensure that the header file specific for the target device is included into the source file.

These header files contain EQU declarations for all byte or multi-byte sized registers and #define macros for named bits within byte registers.

4.3.6.5   SYMBOLIC LABELS

A label is a symbolic alias which is assigned a value equal to the current address within the current psect. Labels are not assigned a value until link time.

A label definition consists of any valid assembly identifier and optionally followed by a *colon*, :. The definition may appear on a line by itself or be positioned before a statement. Here are two examples of legitimate labels interspersed with assembly code.

```
frank:
    MOVLW           1
    GOTO            fin
simon44:   CLRF     _input
```

Here, the label frank will ultimately be assigned the address of the MOVLW instruction, and simon44 the address of the CLRF instruction. Regardless of how they are defined, the assembler list file produced by the assembler will always show labels on a line by themselves.

Note that the colon following the label is optional, but is recommended. Symbols which are not interpreted in any other way are assumed to be labels. Mistyped assembler instructions can sometimes be treated as labels without an error message being issued. Thus the code:

```
mistake:
    MOVLW       23h
    MOVWF       37h
    REUTRN              ; oops
```

defines a symbol called REUTRN, which was intended to be the RETURN instruction.

Labels may be used (and are preferred) in assembly code rather than using an absolute address. Thus they can be used as the target location for jump-type instructions or to load an address into a register.

Like variables, labels have scope. By default, they may be used anywhere in the module in which they are defined. They may be used by code before their definition. To make a label accessible in other modules, use the GLOBAL directive. See **Section 4.3.9.1 "GLOBAL"** for more information.

4.3.7 Expressions

The operands to instructions and directives are comprised of expressions. Expressions can be made up of numbers, identifiers, strings and operators.

Operators can be unary (one operand, e.g. not) or binary (two operands, e.g. +). The operators allowable in expressions are listed in Table 4-4.

TABLE 4-4: ASPIC OPERATORS

Operator	Purpose	Example
*	Multiplication	MOVLW 4*33,w
+	Addition	BRA $+1
–	Subtraction	DB 5-2
/	Division	MOVLW 100/4
= or eq	Equality	IF inp eq 66
> or gt	Signed greater than	IF inp > 40
>= or ge	Signed greater than or equal to	IF inp ge 66
< or lt	Signed less than	IF inp < 40
<= or le	Signed less than or equal to	IF inp le 66
<> or ne	Signed not equal to	IF inp <> 40
low	Low byte of operand	MOVLW low(inp)
high	High byte of operand	MOVLW high(1008h)
highword	High 16 bits of operand	DW highword(inp)
mod	Modulus	MOVLW 77mod4
&	Bitwise AND	CLRF inp&0ffh
^	Bitwise XOR (exclusive or)	MOVF inp^80,w
|	Bitwise OR	MOVF inp|1,w
not	Bitwise complement	MOVLW not 055h,w
<< or shl	Shift left	DB inp>>8
>> or shr	Shift right	MOVLW inp shr 2,w
rol	Rotate left	DB inp rol 1
ror	Rotate right	DB inp ror 1
float24	24-bit version of real operand	DW float24(3.3)
nul	Tests if macro argument is null	

The usual rules governing the syntax of expressions apply.

The operators listed may all be freely combined in both constant and relocatable expressions. The HI-TECH linker permits relocation of complex expressions, so the results of expressions involving relocatable identifiers may not be resolved until link time.

4.3.8   Program Sections

Program sections, or psects, are simply a section of code or data. They are a way of grouping together parts of a program (via the psect's name) even though the source code may not be physically adjacent in the source file, or even where spread over several modules.

A psect is identified by a name and has several attributes. The PSECT assembler directive is used to define a psect. It takes as arguments a name and an optional *comma*-separated list of flags. See **Section 3.7.1 "Compiler-generated Psects"** for a list of all psects that the code generator defines. **Chapter 5. "Linker"** has more information on the operation of the linker and on options that can be used to control psect placement in memory.

The assembler associates no significance to the name of a psect and the linker is also not aware of which psects are compiler-generated or which are user-defined. Unless defined as abs (absolute), psects are relocatable.

The following is an example showing some executable instructions being placed in the mytext psect, and space being reserved in the mybss psect. Neither of these psects are compiler defined.

```
PSECT mytext,class=CODE,delta=2
adjust:
   GOTO   clear_fred
increment:
   INCF   _fred
PSECT mybss,class=BANK0,space=1
fred:
   DS 2
PSECT mytext,class=CODE,delta=2
clear_fred:
   CLRF   _fred+1
   RETURN
```

Note that even though the two blocks of code in the mytext psect are separated by a block in the mybss psect, the two mytext psect blocks will be contiguous when loaded by the linker. In other words, the INCF _fred instruction will be followed by the clrf instruction in the final output. The actual location in memory of the mytext and mybss psects will be determined by the linker.

Code or data that is not explicitly placed into a psect will become part of the default (unnamed) psect.

4.3.9   Assembler Directives

Assembler directives, or pseudo-ops, are used in a similar way to instruction mnemonics. With the exception of PAGESEL and BANKSEL, these directives do not generate any output, or may generate non-executable output, i.e. data bytes. The directives are listed in Table 4-5, and are detailed below in the following sections.

TABLE 4-5:   ASPIC ASSEMBLER DIRECTIVES

Directive	Purpose
GLOBAL	Make symbols accessible to other modules or allow reference to other modules' symbols
END	End assembly
PSECT	Declare or resume program section
ORG	Set location counter within current psect
EQU	Define symbol value
SET	Define or re-define symbol value

TABLE 4-5: ASPIC ASSEMBLER DIRECTIVES (CONTINUED)

Directive	Purpose
DB	Define constant byte(s)
DW	Define constant word(s)
DS	Reserve storage
DABS	Define absolute storage
IF	Conditional assembly
ELSIF	Alternate conditional assembly
ELSE	Alternate conditional assembly
ENDIF	End conditional assembly
FNADDR	Inform the linker that a function may be indirectly called
FNARG	Inform the linker that evaluation of arguments for one function requires calling another
FNBREAK	Break call graph links
FNCALL	Inform the linker that one function calls another
FNCONF	Supply call graph configuration information for the linker
FNINDIR	Inform the linker that all functions with a particular signature may be indirectly called
FNROOT	Inform the linker that a function is the "root" of a call graph
FNSIZE	Inform the linker of argument and local variable for a function
MACRO	Macro definition
ENDM	End macro definition
LOCAL	Define local tabs
ALIGN	Align output to the specified boundary
BANKSEL	Generate code to select bank of operand
PAGESEL	Generate set/clear instruction to set PCLATH bits for this page
PROCESSOR	Define the particular chip for which this file is to be assembled.
REPT	Repeat a block of code *n* times
IRP	Repeat a block of code with a list
IRPC	Repeat a block of code with a character list
SIGNAT	Define function signature

4.3.9.1 GLOBAL

The GLOBAL directive declares a list of *comma*-separated symbols. If the symbols are defined within the current module, they are made public. If the symbols are not defined in the current module, they are made references to public symbols defined in external modules. Thus to use the same symbol in two modules the GLOBAL directive must be used at least twice: once in the module that defines the symbol to make that symbol public, and again in the module that uses the symbol to link in with the external definition.

For example:

```
GLOBAL  lab1,lab2,lab3
```

4.3.9.2 END

The END directive is optional, but if present should be at the very end of the program. It will terminate the assembly and not even blank lines should follow this directive.

If an expression is supplied as an argument, that expression will be used to define the entry point of the program. This is stored in a start record in the object file produced by the assembler. Whether this is of any use will depend on the linker.

The default runtime startup code defined by the compiler will contain an END directive with a start address. As only one start address can be specified for each project, you normally do not need to define this address, but may use the END directive with no entry point in any file.

For example:

```
END  start_label  ;defines the entry point
```

or

```
END               ;do not define entry point
```

4.3.9.3   PSECT

The PSECT directive declares or resumes a program section. It takes as argument a name and, optionally, a *comma*-separated list of flags. The allowed flags are listed in Table 4-6 and specify attributes of the psect.

Once a psect has been declared it may be resumed later by another PSECT directive, however the flags need not be repeated and will be propagated from the earlier declaration. If two PSECT directives are encountered with contradicting flags, then an error will be generated.

TABLE 4-6:   PSECT FLAGS

Flag	Meaning
abs	Psect is absolute
bit	Psect holds bit objects
class=*name*	Specify class name for psect
delta=*size*	Size of an addressing unit
global	Psect is global (default)
limit=*address*	Upper address limit of psect
local	Psect is not global
ovrld	Psect will overlap same psect in other modules
pure	Psect is to be read-only
reloc=*boundary*	Start psect on specified boundary
size=*max*	Maximum size of psect
space=*area*	Represents area in which psect will reside
with=*psect*	Place psect in the same page as specified psect

Some examples of the use of the PSECT directive follow:

```
PSECT fred
PSECT bill,size=100h,global
PSECT joh,abs,ovrld,class=CODE,delta=2
```

4.3.9.3.1   Abs

The abs flag defines the current psect as being absolute, i.e. it is to start at location 0. This does not mean that this module's contribution to the psect will start at 0, since other modules may contribute to the same psect. See also **Section 4.3.9.3.8 "Ovrld"**.

An abs-flagged psect is not relocatable and an error will result if a linker option is issued that attempts to place such a psect at any location.

4.3.9.3.2   Bit

The bit flag specifies that a psect holds objects that are 1 bit long. Such psects will have a scale value of 8 to indicate that there are 8 addressable units to each byte of storage and all addresses associated with this psect will be bit address, not byte addresses. The scale value is indicated in the map file, see **Section 5.4 "Map Files"**.

4.3.9.3.3 Class

The `class` flag specifies a corresponding linker class name for this psect. A class is a range of addresses in which psects may be placed.

Class names are used to allow local psects to be located at link time, since they cannot always be referred to by their own name in a `-P` linker option (as would be the case if there are more than one local psect with the same name).

Class names are also useful where psects need only be positioned anywhere within a range of addresses rather than at a specific address. The association of a class with a psect that you have defined typically means that you do not need to supply a custom linker option to place it in memory.

See **Section 5.2.1 "-Aclass =low-high,..."** for information on how linker classes are defined.

4.3.9.3.4 Delta

The `delta` flag defines the size of the addressing unit. In other words, the number of data bytes which are associated with each address.

With PIC Mid-Range and baseline devices, the program memory space is word addressable, hence psects in this space must use a delta of 2. That is to say, each address in program memory requires 2 bytes of data in the HEX file to define their contents. Thus addresses in the HEX file will not match addresses in the program memory.

The data memory space on these devices is byte addressable, hence psects in this space must use a delta of 1. This is the default delta value.

The redefinition of a psect with conflicting delta values can lead to phase errors being issued by the assembler.

4.3.9.3.5 Global

A psect defined as `global` will be combined with other `global` psects with the same name at link time. Psects are grouped from all modules being linked.

Psects are considered `global` by default, unless the `local` flag is used.

4.3.9.3.6 Limit

The `limit` flag specifies a limit on the highest address to which a psect may extend. If this limit is exceeded when it is positioned in memory, an error will be generated.

4.3.9.3.7 Local

A psect defined as `local` will not be combined with other `local` psects from other modules at link time, even if there are others with the same name. Where there are two `local` psects in the one module, they reference the same psect. A `local` psect may not have the same name as any `global` psect, even one in another module.

4.3.9.3.8 Ovrld

A psect defined as `ovrld` will have the contribution from each module overlaid, rather than concatenated at link time. This flag in combination with the `abs` flag (see **Section 4.3.9.3.1 "Abs"**) defines a truly absolute psect, i.e. a psect within which any symbols defined are absolute.

4.3.9.3.9 Pure

The `pure` flag instructs the linker that this psect will not be modified at runtime and may therefore, for example, be placed in ROM. This flag is of limited usefulness since it depends on the linker and target system enforcing it.

4.3.9.3.10   Reloc

The `reloc` flag allows specification of a requirement for alignment of the psect on a particular boundary. For example the flag `reloc=100h` would specify that this psect must start on an address that is a multiple of 100h.

4.3.9.3.11   Size

The `size` flag allows a maximum size to be specified for the psect, e.g. `size=100h`. This will be checked by the linker after psects have been combined from all modules.

4.3.9.3.12   Space

The `space` flag is used to differentiate areas of memory which have overlapping addresses, but which are distinct. Psects which are positioned in program memory and data memory have a different `space` value to indicate that the program space address 0, for example, is a different location to the data memory address 0.

On all PIC devices, program memory uses a space value of 0, and data space memory uses a space of 1.

Devices which have a banked data space do not use different space values to identify each bank. A full address which includes the bank number is used for objects in this space and so each location can be uniquely identified. For example a device with a bank size of 0x80 bytes will uses address 0 to 0x7F to represent objects in bank 0, and then addresses 0x80 to 0xFF to represent objects in bank 1, etc.

4.3.9.3.13   With

The `with` flag allows a psect to be placed in the same page with another psect. For example the flag `with=text` will specify that this psect should be placed in the same page as the `text` psect.

The term *withtotal* refers to the sum of the size of each psect that is placed "with" other psects.

4.3.9.4   ORG

The `ORG` directive changes the value of the location counter within the current psect. This means that the addresses set with `ORG` are relative to the base address of the psect, which is not determined until link time.

> **Note:**   The much-abused `ORG` directive does *not* move the location counter to the absolute address you specify. Only if the psect in which this directive is placed is absolute and overlaid will the location counter be moved to the address specified. To place objects at a particular address, place them in a psect of their own and link this at the required address using the linkers `-P` option, see **Section 5.2.18 "-Pspec"**. The `ORG` directive is not commonly required in programs.

The argument to `ORG` must be either an absolute value, or a value referencing the current psect. In either case the current location counter is set to the value determined by the argument. It is not possible to move the location counter backward. For example:

```
ORG 100h
```

will move the location counter to the beginning of the current psect plus 100h. The actual location will not be known until link time.

In order to use the `ORG` directive to set the location counter to an absolute value, the directive must be used from within an absolute, overlaid psect. For example:

```
PSECT absdata,abs,ovrld
    ORG 50h
  ;this is guaranteed to reside at address 50h
```

4.3.9.5 EQU

This pseudo-op defines a symbol and equates its value to an expression. For example

```
thomas EQU 123h
```

The identifier `thomas` will be given the value 123h. `EQU` is legal only when the symbol has not previously been defined. See also **Section 4.3.9.6 "SET"** which allows for redefinition of values.

This directive performs a similar function to the preprocessor's `#define` directive, see **Section 3.10.1 "Preprocessor Directives"**.

4.3.9.6 SET

This pseudo-op is equivalent to `EQU` (**Section 4.3.9.5 "EQU"**) except that allows a symbol to be re-defined without error. For example:

```
thomas SET 0h
```

This directive performs a similar function to the preprocessor's `#define` directive, see **Section 3.10.1 "Preprocessor Directives"**.

4.3.9.7 DB

The `DB` directive is used to initialize storage as bytes. The argument is a *comma*-separated list of expressions, each of which will be assembled into one byte and assembled into consecutive memory locations.

Examples:

```
alabel: DB  'X',1,2,3,4,
```

Note that because the size of an address unit in the program memory is 2 bytes (see **Section 4.3.9.3.4 "Delta"**), the `DB` pseudo-op will initialise a word with the upper byte set to zero. So the above example will define bytes padded to the following words.

```
0058 0001 0002 0003 0004
```

4.3.9.8 DW

The `DW` directive operates in a similar fashion to `DB`, except that it assembles expressions into words. Example:

```
DW -1, 3664h, 'A''
```

4.3.9.9 DS

This directive reserves, but does not initialize, memory locations. The single argument is the number of bytes to be reserved.

This directive is typically used to reserve memory location for RAM-based objects in the data memory. If used in a psect linked into the program memory, it will move the location counter, but not place anything in the HEX file output. Note that because the size of an address unit in the program memory is 2 bytes (see **Section 4.3.9.3.4 "Delta"**), the `DS` pseudo-op will actually reserve an entire word.

A variable is typically defined by using a label and then the `DS` directive to reserve locations at the label location.

Examples:

```
alabel: DS 23    ;Reserve 23 bytes of memory
xlabel: DS 2+3   ;Reserve 5 bytes of memory
```

4.3.9.10   DABS

This directive allows one or more bytes of memory to be reserved at the specified address. The general form of the directive is:

```
DABS memorySpace, address, bytes
```

where *memorySpace* is a number representing the memory space in which the reservation will take place, *address* is the address at which the reservation will take place, and *bytes* is the number of bytes that is to be reserved.

This directive differs to the DS directive in that it can be used to reserve memory at any location, not just within the current psect. Indeed, these directives can be placed anywhere in the assembly code and do not contribute to the currently selected psect in any way.

The memory space number is the same as the number specified with the space flag option to psects (see **Section 4.3.9.3.12 "Space"**).

The code generator issues a DABS directive for every user-defined absolute C variable, or for any variables that have been allocated an address by the code generator.

The linker reads this DABS-related information from object files and will ensure that the reserved address are not used for other memory placement.

4.3.9.11   FNADDR

This directive tells the linker that a function has its address taken, and thus could be called indirectly through a function pointer. For example

```
FNADDR _func1
```

tells the linker that func1() has its address taken.

This directive is obsolete and should not be used.

4.3.9.12   FNARG

The directive

```
FNARG fun1,fun2
```

tells the linker that evaluation of the arguments to function fun1 involves a call to fun2, thus the memory argument allocated for the two functions should not overlap. For example, the C function calls

```
fred(var1, bill(), 2);
```

will generate the assembler directive

```
FNARG _fred,_bill
```

thereby telling the linker that bill() is called while evaluating the arguments for a call to fred().

This directive is obsolete and should not be used.

4.3.9.13   FNBREAK

This directive is used to break links in the call graph information. The form of this directive is as follows:

```
FNBREAK fun1,fun2
```

and is automatically generated when the interrupt_level pragma is used. It states that any calls to fun1 in trees other than the one rooted at fun2 should not be considered when checking for functions that appear in multiple call graphs. fun2 is typically intlevel1 in compiler-generated code when an interrupt function is used.

This directive is obsolete and should not be used.

4.3.9.14 FNCALL

This directive takes the form:

```
FNCALL fun1,fun2
```

The FNCALL directive is used by the code generator when performing call graph analysis. The information is passed to the code generator by the driver after scanning object files being compiled. See **Section 3.9.4 "Interaction between Assembly and C Code"** for how this mechanism works.

4.3.9.15 FNCONF

The FNCONF directive is used to supply the linker with configuration information for a call graph. FNCONF is written as follows:

```
FNCONF psect,auto,args
```

where psect is the psect containing the call graph, auto is the prefix on all auto variable symbol names and args is the prefix on all function argument symbol names. This directive normally appears in only one place: the runtime startup code used by C compiler generated code. For the HI-TECH C PRO for PIC10/12/16 MCU Family the startup routine will include the directive:

```
FNCONF rbss,??,?
```

telling the linker that the call graph is in the rbss psect, auto variable blocks start with ?? and function argument blocks start with ?.

This directive is obsolete and should not be used.

4.3.9.16 FNINDIR

This directive tells the linker that a function performs an indirect call to another function with a particular signature (see the SIGNAT directive). The linker must assume worst case that the function could call any other function which has the same signature and has had its address taken (see the FNADDR directive). For example, if a function called fred() performs an indirect call to a function with signature 8249, the compiler will produce the directive:

```
FNINDIR _fred,8249
```
This directive is obsolete and should not be used.

4.3.9.17 FNSIZE

The FNSIZE directive is used by the code generator when building the call graph and assigning addresses to the variable and argument areas. The information is passed to the code generator by the driver after scanning object files being compiled. See **Section 3.9.4 "Interaction between Assembly and C Code"** for how this mechanism works.

This directive takes the form:

```
FNSIZE func,local,args
```

The named function has a local variable area and argument area as specified, for example

```
FNSIZE _fred, 10, 5
```

means the routine _fred has 10 bytes of local variables and 5 bytes of arguments.

4.3.9.18 FNROOT

This directive tells the assembler that a function is a root function and thus forms the root of a call graph. It could either be the C `main()` function or an interrupt function. For example, the C main module produce the directive:

```
FNROOT _main
```

This directive is obsolete and should not be used.

4.3.9.19 IF, ELSIF, ELSE AND ENDIF

These directives implement conditional assembly. The argument to `IF` and `ELSIF` should be an absolute expression. If it is non-zero, then the code following it up to the next matching `ELSE`, `ELSIF` or `ENDIF` will be assembled. If the expression is zero then the code up to the next matching `ELSE` or `ENDIF` will be skipped. These directives do not implement a runtime conditional statement in the same way that the C statement `if()` does; they are evaluated at compile time.

At an `ELSE` the sense of the conditional compilation will be inverted, while an `ENDIF` will terminate the conditional assembly block.

For example:

```
IF ABC
    GOTO aardvark
ELSIF DEF
    GOTO denver
ELSE
    GOTO grapes
ENDIF
```

In this example, if `ABC` is non-zero, the first `GOTO` instruction will be assembled but not the second or third. If `ABC` is zero and `DEF` is non-zero, the second `GOTO` instruction will be assembled but the first and third will not. If both `ABC` and `DEF` are zero, the third `GOTO` instruction will be assembled. Note in the above example, only one `GOTO` instruction will appear in the output; which one will be determined by the values assigned to `ABC` and `DEF`.

Conditional assembly blocks may be nested.

4.3.9.20 MACRO AND ENDM

These directives provide for the definition of assembly macros, optionally with arguments. See **Section 4.3.9.5 "EQU"** for simple association of a value with an identifier, or **Section 3.10.1 "Preprocessor Directives"** for the preprocessor's `#define` macro directive, which can also work with arguments.

The `MACRO` directive should be preceded by the macro name and optionally followed by a *comma*-separated list of formal arguments. When the macro is used, the macro name should be used in the same manner as a machine opcode, followed by a list of arguments to be substituted for the formal parameters.

For example:

```
;macro: movlf
;args:  arg1 - the literal value to load
;       arg2 - the NAME of the source variable
;descr: Move a literal value into a nominated file register

movlf   MACRO   arg1,arg2
    MOVLW  arg1
    MOVWF  arg2 mod 080h
ENDM
```

When used, this macro will expand to the 2 instructions in the body of the macro, with the formal parameters substituted by the arguments. Thus:

```
movlf 2,tempvar
```

expands to:

```
MOVLW  2
MOVWF  tempvar mod 080h
```

The `&` character can be used to permit the concatenation of macro arguments with other text, but is removed in the actual expansion. For example:

```
loadPort MACRO port, value
    MOVLW value
    MOVWF PORT&port
ENDM
```

will load `PORTA` if `port` is `A` when called, etc.

A comment may be suppressed within the expansion of a macro (thus saving space in the macro storage) by opening the comment with a double semicolon, `;;`.

When invoking a macro, the argument list must be *comma*-separated. If it is desired to include a *comma* (or other delimiter such as a space) in an argument then angle brackets `<` and `>` may be used to quote

If an argument is preceded by a percent sign, `%`, that argument will be evaluated as an expression and passed as a decimal number, rather than as a string. This is useful if evaluation of the argument inside the macro body would yield a different result.

The `nul` operator may be used within a macro to test a macro argument, for example:

```
IF nul    arg3  ; argument was not supplied.
    ...
ELSE            ; argument was supplied
    ...
ENDIF
```

See **Section 4.3.9.21 "LOCAL"** for use of unique local labels within macros.

By default, the assembly list file will show macro in an unexpanded format, i.e. as the macro was invoked. Expansion of the macro in the listing file can be shown by using the `EXPAND` assembler control, see **Section 4.3.10.2 "EXPAND"**.

4.3.9.21   LOCAL

The `LOCAL` directive allows unique labels to be defined for each expansion of a given macro. Any symbols listed after the `LOCAL` directive will have a unique assembler generated symbol substituted for them when the macro is expanded. For example:

```
down MACRO count
    LOCAL  more
    more:  DECFSZ     count
           GOTO       more
ENDM
```

when expanded will include a unique assembler generated label in place of `more`. For example:

```
down foobar
```

expands to:

```
??0001 DECFSZ    foobar
       GOTO      ??0001
```

If invoked a second time, the label `more` would expand to `??0002` and multiply defined symbol errors will be averted.

4.3.9.22 ALIGN

The `ALIGN` directive aligns whatever is following, data storage or code etc., to the specified offset boundary within the current psect. The boundary is specified as a number of bytes following the directive.

For example, to align output to a 2 byte (even) address within a psect, the following could be used.

```
ALIGN 2
```

Note that what follows will only begin on an even absolute address if the psect begins on an even address, i.e. alignment is done within the current psect. See **Section 4.3.9.3.10 "Reloc"** for psect alignment.

The `ALIGN` directive can also be used to ensure that a psect's length is a multiple of a certain number. For example, if the above `ALIGN` directive was placed at the end of a psect, the psect would have a length that was always an even number of bytes long.

4.3.9.23 REPT

The `REPT` directive temporarily defines an unnamed macro, then expands it a number of times as determined by its argument.

For example:

```
REPT 3
    ADDWF       fred,w
ENDM
```

will expand to:

```
    ADDWF       fred,w
    ADDWF       fred,w
    ADDWF       fred,w
```

See also **Section 4.3.9.24 "IRP and IRPC"**.

4.3.9.24 IRP AND IRPC

The `IRP` and `IRPC` directives operate in a similar way to `REPT`, however instead of repeating the block a fixed number of times, it is repeated once for each member of an argument list.

In the case of `IRP`, the list is a conventional macro argument list; in the case or `IRPC`, it is each character in one argument. For each repetition the argument is substituted for one formal parameter.

For example:

```
IRP number,4865h,6C6Ch,6F00h
  DW number
ENDM
```

would expand to:

```
  DW 4865h
  DW 6C6Ch
  DW 6F00h
```

Note that you can use local labels and angle brackets in the same manner as with conventional macros.

The IRPC directive is similar, except it substitutes one character at a time from a string of non-space characters.

For example:

```
IRPC char,ABC
  DB 'char'
ENDM
```

will expand to:

```
  DB 'A'
  DB 'B'
  DB 'C'
```

4.3.9.25 BANKSEL

This directive can be used to generate code to select the bank of the operand. The operand should be the symbol or address of an object that resides in the data memory.

Depending on the target device, the generated code will either contain one or more bit instructions to set/clear bits in the appropriate register, or use a MOVLB instruction in the case of enhanced Mid-Range PIC devices. In case this directive expands to more than one instruction, it should not immediately follow a BTFS*X* instruction.

For example:

```
MOVLW    20
BANKSEL  (_foobar)        ; select bank for next file instruction
MOVWF    _foobar&07fh     ; write data and mask address
```

4.3.9.26 PAGESEL

This directive can be used to generate code to select the current page, i.e. the page which contains this directive.

Depending on the target device, the generated code will either contain one or more bit instructions to set/clear bits in the appropriate register, or use a MOVLP instruction in the case of enhanced Mid-Range PIC devices. In case this directive expands to more than one instruction, it should not immediately follow a BTFS*X* instruction.

For example:

```
CALL     _getInput
PAGESEL                   ; select this page
```

4.3.9.27 PROCESSOR

The output of the assembler may vary depending on the target device. The device name is typically set using the --CHIP option to the command-line driver PICC, see **Section 2.7.20 "--CHIP: Define Processor"**, or using the assembler's --CHIP option, see **Section 4.2.15 "--CHIP: Specify Device Name"**, but can also be set with this directive, e.g.

```
PROCESSOR 16F877
```

This directive will override any processor selected by a command-line option.

4.3.9.28   SIGNAT

This directive is used to associate a 16-bit signature value with a label. At link time the linker checks that all signatures defined for a particular label are the same and produces an error if they are not. The SIGNAT directive is used by HI-TECH C to enforce link time checking of C function prototypes and calling conventions.

Use the SIGNAT directive if you want to write assembly language routines which are called from C. For example:

```
SIGNAT _fred,8192
```

will associate the signature value 8192 with the symbol _fred. If a different signature value for _fred is present in any object file, the linker will report an error.

The easiest way to determine the correct signature value for a routine is to write a C routine with the same prototype as the assembly routine and check the signature value determined by the code generator. This will be shown in the assembly list file, see **Section 2.7.17 "--ASMLIST: Generate Assembler List Files"** and **Section 4.4 "Assembly List Files"**.

4.3.10   Assembler Controls

Assembler controls may be included in the assembler source to control assembler operation. These keywords have no significance anywhere else in the program. The control is invoked by the directive OPT followed by the control name. Some keywords are followed by one or more arguments. For example:

```
OPT EXPAND
```

A list of keywords is given in Table 4-7, and each is described further below.

TABLE 4-7:   ASPIC ASSEMBLER CONTROLS

Control	Meaning	Format
COND*	Include conditional code in the listing	COND
EXPAND	Expand macros in the listing output	EXPAND
INCLUDE	Textually include another source file	INCLUDE < *pathname* >
LIST*	Define options for listing output	LIST [< *listopt* >, ..., < *listopt* >]
NOCOND	Leave conditional code out of the listing	NOCOND
NOEXPAND*	Disable macro expansion	NOEXPAND
NOLIST	Disable listing output	NOLIST
NOXREF	Disable generation of cross reference file	NOXREF
PAGE	Start a new page in the listing output	PAGE
SPACE	Add blank lines to listing	SPACE 3
SUBTITLE	Specify the subtitle of the program	SUBTITLE "< *subtitle* >"
TITLE	Specify the title of the program	TITLE "< *title* >"
XREF	Enable cross reference file generation	XREF

Note 1:   The default options are listed with an asterisk (*)

4.3.10.1   COND

Any conditional code will be included in the listing output. See also the NOCOND control in **Section 4.3.10.5 "NOCOND"**.

4.3.10.2   EXPAND

When EXPAND is in effect, the code generated by macro expansions will appear in the listing output. See also the NOEXPAND control in **Section 4.3.10.6 "NOEXPAND"**.

4.3.10.3   INCLUDE

This control causes the file specified by *pathname* to be textually included at that point in the assembly file. The INCLUDE control must be the last control keyword on the line, for example:

```
OPT INCLUDE "options.h"
```

The driver does not pass any search paths to the assembler, so if the include file is not located in the working directory, the pathname must specify the exact location.

See also the driver option -P in **Section 2.7.11 "-P: Preprocess Assembly Files"** which forces the C preprocessor to preprocess assembly file, thus allowing use of preprocessor directives, such as #include (see **Section 3.10.1 "Preprocessor Directives"**).

4.3.10.4   LIST

If the listing was previously turned off using the NOLIST control, the LIST control on its own will turn the listing on.

Alternatively, the LIST control may includes options to control the assembly and the listing. The options are listed in Table 4-8.

TABLE 4-8:   LIST CONTROL OPTIONS

List Option	Default	Description
c= *nnn*	80	Set the page (i.e. column) width.
n= *nnn*	59	Set the page length.
t= *ON|OFF*	OFF	Truncate listing output lines. The default wraps lines.
p=< *processor* >	n/a	Set the processor type.
r=< *radix* >	HEX	Set the default radix to HEX, dec or oct.
x= *ON|OFF*	OFF	Turn macro expansion on or off.

See also the NOLIST control in **Section 4.3.10.7 "NOLIST"**.

4.3.10.5   NOCOND

Using this control will prevent conditional code from being included in the assembly list file output. See also the COND control in **Section 4.3.10.1 "COND"**.

4.3.10.6   NOEXPAND

The NOEXPAND control disables macro expansion in the assembly list file. The macro call will be listed instead. See also the EXPAND control in **Section 4.3.10.2 "EXPAND"**. Assembly macro are discussed in **Section 4.3.9.20 "MACRO and ENDM"**.

4.3.10.7   NOLIST

This control turns the listing output off from this point onward. See also the LIST control in **Section 4.3.10.4 "LIST"**.

4.3.10.8 NOXREF

The `NOXREF` control will disable generation of the *raw* cross reference file. See also the `XREF` control in **Section 4.3.10.13 "XREF"**.

4.3.10.9 PAGE

The `PAGE` control causes a new page to be started in the listing output. A Control-L (form feed) character will also cause a new page when encountered in the source.

4.3.10.10 SPACE

The `SPACE` control will place a number of blank lines in the listing output as specified by its parameter.

4.3.10.11 SUBTITLE

The `SUBTITLE` control defines a subtitle to appear at the top of every listing page, but under the title. The string should be enclosed in single or double quotes. See also the `TITLE` control in **Section 4.3.10.12 "TITLE"**.

4.3.10.12 TITLE

This control keyword defines a title to appear at the top of every listing page. The string should be enclosed in single or double quotes. See also the `SUBTITLE` control in **Section 4.3.10.11 "SUBTITLE"**.

4.3.10.13 XREF

The `XREF` control is equivalent to the driver command line option `--CR` (see **Section 2.7.23 "--CR: Generate Cross Reference Listing"**). It causes the assembler to produce a raw cross reference file. The utility `CREF` should be used to actually generate the formatted cross-reference listing.

4.4   ASSEMBLY LIST FILES

The assembler will produce an assembly list file if instructed. The PICC driver option `--ASMLIST` is typically used to request generation of such a file, see **Section 2.7.17 "--ASMLIST: Generate Assembler List Files"**.

The assembly list file shows the assembly output produced by the compiler for both C and assembly source code. If the assembler optimizers are enabled, the assembly output may be different to assembly source code and so is still useful for assembly programming.

The list file is in a human readable form and cannot take any further part in the compilation sequence. It differs from an assembly output file in that it contains address and op-code data. In addition, the assembler optimizer simplifies some expressions and removes some assembler directives from the listing file for clarity, although these directives are included in the true assembly output files. If you are using the assembly list file to look at the code produced by the compiler, you may wish to turn off the assembler optimizer so that all the compiler-generated directives are shown in this file. Re-enable the optimizer when continuing development. **Section 2.7.42 "--OPT: Invoke Compiler Optimizations"** gives more information on controlling the optimizers.

Provided the link stage has successfully concluded, the listing file will be updated by the linker so that it contains absolute addresses and symbol values. Thus you may use the assembler list file to determine the position of, and exact op codes of, instructions.

There is one assembly list file produce by the assembler for each assembly file passed to it, and so there will be one file produced for all the C source code in a project, including p-code based library code. This file will also contains some of the C initialization

that forms part of the runtime startup code. There will also be one file produced for each assembly source file. There is typically at least one assembly file in each project, that containing some of the runtime startup file, typically called `startup.as`.

4.4.1   General Format

The format of the main listing has the form as shown in **Section Figure 4-1: "General form of assembly listing file"**.

The line numbers purely relate to the assembly list file and are not associated with the lines numbers in the C or assembly source files. Any assembly that begins with a semi-colon indicates it is a comment added by the code generator. Such comments contain either the original source code which corresponds to the generated assembly, or is a comment inserted by the code generator to explain some action taken.

Before the output for each function there is detailed information regarding that function summarized by the code generator. This information relates to register usage, local variable information, functions called and the calling function.

FIGURE 4-1:   GENERAL FORM OF ASSEMBLY LISTING FILE
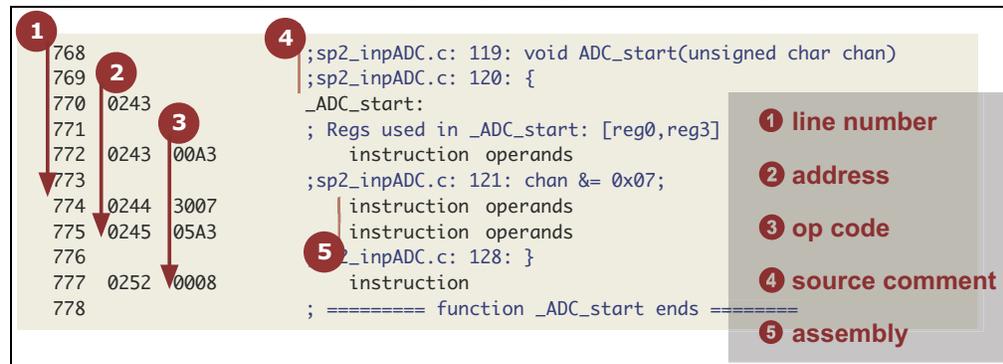
4.4.2   Pointer Reference Graph

Other important information contained in the assembly list file is the pointer reference graph (look for *Pointer list with targets:* in the list file). This is a list of each and every pointer contained in the program and each target the pointer can reference through the program. The size and type of each target is indicated as well as the size and type of the pointer variable itself.

For example, the following shows a pointer called `task_tmr` in the C code, and which is local to the function `timer_intr()`. It is a pointer to an `unsigned int` and it is one byte wide. There is only one target to this pointer and it is the member `timer_count` in the structure called `task`. This target variable resides in the BANK0 class and is two bytes wide.

```
timer_intr@task_tmr   PTR unsigned int  size(1); Largest target is 2
                         -> task.timer_count(BANK0[2]),
```

The pointer reference graph shows both pointer to data objects and pointers to functions.

4.4.3 Call Graph

The other important information block in the assembly list file is the call graph (look for *Call graph:* in the list file). This is produced for target devices that use a compiled stack to facilitate local variables, such as function parameters and `auto` variables. See **Section 3.4.2 "Compiled Stack Operation"** for more detailed information on compiled stack operation.

The call graph in the list file shows the information collated and interpreted by the code generator, which is primarily used to allow overlapping of functions' APBs. The following information can be obtained from studying the call graph.

- The functions in the program that are "root" nodes marking the top of a call tree, and which are called spontaneously
- The functions that the linker deemed were called, or may have been called, during program execution
- The program's hierarchy of function calls
- The size of the auto and parameter areas within each function's APB
- The offset of each function's APB within the compiled stack
- The estimated call tree depth.

These features are discussed below.

A typical call graph may look that shown inFigure 4-2.

FIGURE 4-2:   CALL GRAPH FORM


```
Call graph:                        Base Space Used Autos Args Refs Density
_main                                               10    0   24   0.00
                            4 COMMO    6
                           16 BANK0    4
               _rv
               _rvx
               _rvy
_rvx                                                 0    2    9   0.00
                            8 BANK0    2
                   _rv2
_rvy                                                 0    2    3   0.00
                            0 BANK0    2
_rv                                                  8    4   12   0.00
                            0 COMMO    4
                            8 BANK0    8
                   _rv2
_rv2                                                 4    4    6   0.00
                            0 BANK0    8
Estimated maximum call depth 2
```

The graph starts with the function `main()`. Note that the function name will always be shown in the assembly form, thus the function `main()` appears as the symbol `_main`. `main()` is always a root of a call tree. Interrupt functions will form separate trees.

All the functions that `main()` calls, or may call, are shown below. These have been grouped in the orange box in the figure. A function's inclusion into the call graph does not imply the function was actually called, but there is a possibility that the function was called. For example, code such as:

```
int test(int a) {
  if(a)
    foo();
  else
    bar();
}
```

will list `foo()` and `bar()` under `test()`, as either may be called. If `a` is always true, then the function `bar()` will never be called even though it appears in the call graph.

In addition to these functions there is information relating to the memory allocated in the compiled stack for `main()`. This memory will be used for `auto`, temporary and parameter variables defined in `main()`. The only difference between an `auto` and temporary variable is that `auto` variables are defined by the programmer, and temporaries are defined by the compiler, but both behave in the same way.

In the orange box for `main()` you can see that it defines 10 `auto` and temporary variable. It defines no parameters (`main()` never has parameters). There is a total of 24 references in the assembly code to local objects in `main()`.

Rather than the compiled stack being one memory allocation in one memory space, it can have components placed in multiple memory spaces to utilize all available memory of the target device. This break down is shown under the memory summary line for each function. In this example, it shows that some of the local objects for `main()` are placed in the common memory, but others are placed in bank 0 data RAM.

The *Used* column indicates how many bytes of memory are used by each section of the compiled stack and the *Space* column indicates in which space that has been placed. The *Base* value indicates the offset that block has in the respective section of the compiled stack. For example, the figure tells us `main()` has 6 bytes of memory allocated at an offset of 4 in the compiled stack section that lives in common memory. It also has 4 bytes of memory allocated in bank 0 memory at an offset of 16 in the bank 0 compiled stack component.

Below the information for `main()` (outside the orange box) you will see the same information repeated for the functions that `main()` called, viz. `rv()`, `rvx()` and `rvy()`. Indentation is used to indicate the maximum depth that function reaches in the call graph. The arrows in the figure highlight this indentation.

After each tree in the call graph, there is an indication of the maximum call (stack) depth that might be realized by that tree. This may be used as a guide to the stack usage of the program. No definitive value can be given for the program's total stack usage for several reasons:

- Certain parts of the call tree may never be reached, reducing that tree's stack usage.
- The contribution of interrupt (or other) trees to the `main()` tree cannot be determined as the point in `main`'s call tree at which the interrupt (or other function invocation) will occur cannot be known;
- The assembler optimizer may have replaced function calls with jumps to functions, reducing that tree's stack usage.
- The assembler's procedural abstraction optimizations may have added in calls to abstracted routines. (Checks are made to ensure this does not exceed the maximum stack depth.)

The code generator also produces a warning if the maximum stack depth appears to have been exceeded. For the above reasons, this warning, too, is intended to be a only a guide to potential stack problems.

Chapter 5. Linker

5.1 INTRODUCTION

This chapter describes the theory behind, and the usage of, the linker.

The application name of the linker is HLINK. In most instances it will not be necessary to invoke the linker directly, as the compiler driver, PICC, will automatically execute the linker with all necessary arguments. Using the linker directly is not simple, and should be attempted only by those with a sound knowledge of the compiler and linking in general. The compiler often makes assumptions about the way in which the program will be linked. If the psects are not linked correctly, code failure may result.

If it is absolutely necessary to use the linker directly, the best way to start is to copy the linker arguments constructed by the compiler driver, and modify them as appropriate. This will ensure that the necessary startup module and arguments are present.

5.2 OPERATION

A command to the linker takes the following form:

```
hlink [options] files
```

The *options* are zero or more linker options, each of which modifies the behavior of the linker in some way. The *files* is one or more object files, and zero or more object code library names (.lib extension). P-code libraries (.lpp extension) are always passed to the code generator application and cannot be passed to the linker.

The options recognized by the linker are listed in Table 5-1 and discussed in the following paragraphs.

TABLE 5-1:   LINKER COMMAND-LINE OPTIONS

Option	Effect
-8	Use 8086 style segment:offset address form
-A*class=low-high* , . . .	Specify address ranges for a class
-C*x*	Call graph options
-C*psect=class*	Specify a class name for a global psect
-C*baseaddr*	Produce binary output file based at *baseaddr*
-D*class=delta*	Specify a class delta value
-D*symfile*	Produce old-style symbol file
-E*errfile*	Write error messages to *errfile*
-F	Produce .obj file with only symbol records
-G *spec*	Specify calculation for segment selectors
-H *symfile*	Generate symbol file
-H+ *symfile*	Generate enhanced symbol file
-I	Ignore undefined symbols
-J *num*	Set maximum number of errors before aborting
-K	Prevent overlaying function parameter and auto areas
-L	Preserve relocation items in .obj file
-LM	Preserve segment relocation items in .obj file

TABLE 5-1:   LINKER COMMAND-LINE OPTIONS (CONTINUED)

Option	Effect
-N	Sort symbol table in map file by address order
-Nc	Sort symbol table in map file by class address order
-Ns	Sort symbol table in map file by space address order
-M*mapfile*	Generate a link map in the named file
-O*outfile*	Specify name of output file
-P*spec*	Specify psect addresses and ordering
-Q*processor*	Specify the processor type (for cosmetic reasons only)
-S	Inhibit listing of symbols in symbol file
-S*class=limit[,bound]*	Specify address limit, and start boundary for a class of psects
-U*symbol*	Pre-enter symbol in table as undefined
-V*avmap*	Use file *avmap* to generate an *Avocet* format symbol file
-W*warnlev*	Set warning level (-9 to 9)
-W*width*	Set map file width (>=10)
-X	Remove any local symbols from the symbol file
-Z	Remove trivial local symbols from the symbol file
--DISL=*list*	Specify disabled messages
--EDF=*path*	Specify message file location
--EMAX=*number*	Specify maximum number of errors
--NORLF	Do not relocate list file
--VER	Print version number and stop

If the standard input is a file then this file is assumed to contain the command-line argument. Lines may be broken by leaving a *backslash* \ at the end of the preceding line. In this fashion, HLINK commands of almost unlimited length may be issued. For example a link command file called x.lnk and containing the following text:

```
-Z -OX.OBJ -MX.MAP \
-Ptext=0,data=0/,bss,nvram=bss/. \
X.OBJ Y.OBJ Z.OBJ
```

may be passed to the linker by one of the following:

```
hlink @x.lnk
hlink < x.lnk
```

Several linker options require memory addresses or sizes to be specified. The syntax for all these is similar. By default, the number will be interpreted as a decimal value. To force interpretation as a HEX number, a trailing H, or h, should be added, e.g. 765FH will be treated as a HEX number.

5.2.1   **-A***class =low-high*,...

Normally psects are linked according to the information given to a -P option (see **Section 5.2.18 "-Pspec"**) but sometimes it is desirable to have a class of psects linked into more than one non-contiguous address range. This option allows a number of address ranges to be specified as a class. For example:

```
-ACODE=1020h-7FFEh,8000h-BFFEh
```

specifies that psects in the class CODE are to be linked into the given address ranges, unless they are specifically linked otherwise.

Where there are a number of identical, contiguous address ranges, they may be specified with a repeat count following an `x` character. For example:

```
-ACODE=0-0FFFFhx16
```

specifies that there are 16 contiguous ranges, each 64k bytes in size, starting from address zero. Even though the ranges are contiguous, no psect will straddle a 64k boundary, thus this may result in different psect placement to the case where the option

```
-ACODE=0-0FFFFFh
```

had been specified, which does not include boundaries on 64k multiples.

The `-A` option does not specify the memory space associated with the address. Once a psect is allocated to a class, the space value of the psect is then assigned to the class, see **Section 4.3.9.3.12 "Space"**.

5.2.2   -C*x*

This option is now obsolete.

-Cpsect=class

This option will allow a psect to be associated with a specific class. Normally this is not required on the command line since psect classes are specified in object files. See **Section 4.3.9.3.3 "Class"**.

5.2.3   -D*class=delta*

This option allows the delta value for psects that are members of the specified class to be defined. The delta value should be a number and represents the number of bytes per addressable unit of objects within the psects. Most psects do not need this option as they are defined with a delta value. See **Section 4.3.9.3.4 "Delta"**.

5.2.4   -D*symfile*

Use this option to produce an old-style symbol file. An old-style symbol file is an ASCII file, where each line has the link address of the symbol followed by the symbol name.

5.2.5   -E*errfile*

Error messages from the linker are written to the standard error stream. Under DOS there is no convenient way to redirect this to a file (the compiler drivers will redirect standard error if standard output is redirected). This option will make the linker write all error messages to the specified file instead of the screen, which is the default standard error destination.

5.2.6   -F

Normally the linker will produce an object file that contains both program code and data bytes, and symbol information. Sometimes it is desired to produce a symbol-only object file that can be used again in a subsequent linker run to supply symbol values. The `-F` option will suppress data and code bytes from the output file, leaving only the symbol records.

This option can be used when part of one project (i.e. a separate build) is to be shared with other, as might be the case with a bootloader and application. The files for one project are compiled using this linker option to produce a symbol-only object file; this is then linked with the files for the other project.

5.2.7 -G*spec*

When linking programs using segmented, or bank-switched psects, there are two ways the linker can assign segment addresses, or selectors, to each segment. A segment is defined as a contiguous group of psects where each psect in sequence has both its link and load address concatenated with the previous psect in the group. The segment address or selector for the segment is the value derived when a segment type relocation is processed by the linker.

By default the segment selector will be generated by dividing the base load address of the segment by the relocation quantum of the segment, which is based on the `reloc=` flag value given to psects at the assembler level, see **Section 4.3.9.3.10 "Reloc"**. The `-G` option allows an alternate method for calculating the segment selector. The argument to `-G` is a string similar to:

```
A /10h-4h
```

where *A* represents the load address of the segment and `/` represents division. This means "Take the load address of the psect, divide by 10 HEX, then subtract 4". This form can be modified by substituting *N* for *A*, `*` for `/` (to represent multiplication), and adding rather than subtracting a constant. The token *N* is replaced by the ordinal number of the segment, which is allocated by the linker. For example:

```
N*8+4
```

means "take the segment number, multiply by 8 then add 4". The result is the segment selector. This particular example would allocate segment selectors in the sequence 4, 12, 20, ... for the number of segments defined.

The selector of each psect is shown in the map file. See **Section 5.4.2.2 "Psect Information listed by Module"**.

5.2.8 -H*symfile*

This option will instruct the linker to generate a symbol file. The optional argument `symfile` specifies the name of the file to receive the data. The default file name is `l.sym`.

5.2.9 -H+*symfile*

This option will instruct the linker to generate an enhanced symbol file, which provides, in addition to the standard symbol file, class names associated with each symbol and a segments section which lists each class name and the range of memory it occupies. This format is recommended if the code is to be run in conjunction with a debugger. The optional argument *symfile* specifies a file to receive the symbol file. The default file name is `l.sym`.

5.2.10 -I

Usually failure to resolve a reference to an undefined symbol is a fatal error. Use of this option will cause undefined symbols to be treated as warnings instead.

5.2.11 -J*errcount*

The linker will stop processing object files after a certain number of errors (other than warnings). The default number is 10, but the `-J` option allows this to be altered.

5.2.12 -K

For compilers that use a compiled stack, the linker will try and overlay function auto and parameter blocks to reduce the total amount of RAM required. For debugging purposes, this feature can be disabled with this option, however doing so will increase the data memory requirements.

5.2.13 -L

When the linker produces an output file it does not usually preserve any relocation information, since the file is now absolute. In some circumstances a further "relocation" of the program will be done at load time. The -L option will generate in the output file one null relocation record for each relocation record in the input.

5.2.14 -LM

Similar to the above option, this preserves relocation records in the output file, but only segment relocations.

5.2.15 -M*mapfile*

This option causes the linker to generate a link map in the named file, or on the standard output if the file name is omitted. The format of the map file is illustrated in Section **Section 5.4 "Map Files"**.

5.2.16 -N, -Ns and-Nc

By default the symbol table in the map file will be sorted by name. The -N option will cause it to be sorted numerically, based on the value of the symbol. The -Ns and -Nc options work similarly except that the symbols are grouped by either their space value, or class.

5.2.17 -O*outfile*

This option allows specification of an output file name for the linker. The default output file name is l.obj. Use of this option will override the default.

5.2.18 -P*spec*

Psects are linked together and assigned addresses based on information supplied to the linker via -P options. The argument to the -P option consists basically of *comma* -separated sequences thus:

```
-Ppsect =lnkaddr+min/ldaddr+min,psect=lnkaddr/ldaddr,...
```

There are several variations, but essentially each psect is listed with its desired link and load addresses, and a minimum value. All values may be omitted, in which case a default will apply, depending on previous values.

If present, the minimum value, *min*, is preceded by a + sign. It sets a minimum value for the link or load address. The address will be calculated as described below, but if it is less than the minimum then it will be set equal to the minimum.

The link and load addresses are either numbers, or the names of other psects, classes, or special tokens.

If the link address is a negative number, the psect is linked in reverse order with the top of the psect appearing at the specified address minus one. Psects following a negative address will be placed before the first psect in memory.

If a psect's link address is omitted, it will be derived from the top of the previous psect. For example, in the following:

```
-Ptext=100h,data,bss
```

the text psect is linked at 100h (its load address defaults to the same). The data psect will be linked (and loaded) at an address which is 100 HEX plus the length of the text psect, rounded up as necessary if the data psect has a reloc value associated with it (see **Section 4.3.9.3.10 "Reloc"**). Similarly, the bss psect will concatenate with the data psect. Again:

```
-Ptext=-100h,data,bss
```

will link in ascending order `bss`, `data` then `text` with the top of the `text` psect appearing at address 0ffh.

If the load address is omitted entirely, it defaults to the same as the link address. If the *slash* `/` character is supplied, but no address is supplied after it, the load address will concatenate with the previous psect. For example:

```
-Ptext=0,data=0/,bss
```

will cause both `text` and `data` to have a link address of zero; `text` will have a load address of zero, and `data` will have a load address starting after the end of `text`. The `bss` psect will concatenate with `data` in terms of both link and load addresses.

The load address may be replaced with a *dot* character, "`.`". This tells the linker to set the load address of this psect to the same as its link address. The link or load address may also be the name of another (previously linked) psect. This will explicitly concatenate the current psect with the previously specified psect, e.g.

```
-Ptext=0,data=8000h/,bss/. -Pnvram=bss,heap
```

This example shows `text` at zero, `data` linked at 8000h but loaded after `text`; `bss` is linked and loaded at 8000h plus the size of `data`, and `nvram` and `heap` are concatenated with `bss`. Note here the use of two `-P` options. Multiple `-P` options are processed in order.

If `-A` options (see **Section 5.2.1 "-Aclass =low-high,..."**) have been used to specify address ranges for a class then this class name may be used in place of a link or load address, and space will be found in one of the address ranges. For example:

```
-ACODE=8000h-BFFEh,E000h-FFFEh
-Pdata=C000h/CODE
```

This will link `data` at C000h, but find space to load it in the address ranges associated with the `CODE` class. If no sufficiently large space is available in this class, an error will result. Note that in this case the `data` psect will still be assembled into one contiguous block, whereas other psects in the class `CODE` will be distributed into the address ranges wherever they will fit. This means that if there are two or more psects in class `CODE`, they may be intermixed in the address ranges.

Any psects allocated by a `-P` option will have their load address range subtracted from the address ranges associate with classes in the same memory space. This allows a range to be specified with the `-A` option without knowing in advance how much of the lower part of the range, for example, will be required for other psects.

The final link and load address of psects are shown in the map file. See **Section 5.4.2.2 "Psect Information listed by Module"**.

5.2.19   -Q*processor*

This option allows a processor type to be specified. This is purely for information placed in the map file. The argument to this option is a string describing the processor. There are no behavioral changes attributable to the processor type.

5.2.20   -S

This option prevents symbol information relating from being included in the symbol file produced by the linker. Segment information is still included.

5.2.21   -S*class =limit[,bound]*

A class of psects may have an upper address limit associated with it. The following example places a limit on the maximum address of the `CODE` class of psects to one less than 400h.

```
-SCODE=400h
```

Note that to set an upper limit to a psect, this must be set in assembler code using the psect `limit` flag, see **Section 4.3.9.3.6 "Limit"**).

If the bound (boundary) argument is used, the class of psects will start on a multiple of the bound address. This example below places the `FARCODE` class of psects at a multiple of 1000h, but with an upper address limit of 6000h.

```
-SFARCODE=6000h,1000h
```

5.2.22   **-U***symbol*

This option will enter the specified symbol into the linker's symbol table as an undefined symbol. This is useful for linking entirely from libraries, or for linking a module from a library where the ordering has been arranged so that by default a later module will be linked.

5.2.23   **-V***avmap*

To produce an Avocet format symbol file, the linker needs to be given a map file to allow it to map psect names to Avocet memory identifiers. The avmap file will normally be supplied with the compiler, or created automatically by the compiler driver as required.

5.2.24   **-W***num*

The `-W` option can be used to set the warning level, in the range -9 to 9, or the width of the map file, for values of *num* >= 10.

`-W9` will suppress all warning messages. `-W0` is the default. Setting the warning level to -9 (`-W-9`) will give the most comprehensive warning messages.

5.2.25   **-X**

Local symbols can be suppressed from a symbol file with this option. Global symbols will always appear in the symbol file.

5.2.26   **-Z**

Some `local` symbols are compiler generated and not of interest in debugging. This option will suppress from the symbol file all local symbols that have the form of a single alphabetic character, followed by a digit string. The set of letters that can start a trivial symbol is currently "`klfLSu`". The `-Z` option will strip any local symbols starting with one of these letters, and followed by a digit string.

5.2.27   **--DISL=***message numbers* **Disable Messages**

This option is mainly used by the command-line driver, PICC, to disable particular message numbers. It takes a *comma*-separate list of message numbers that will be disabled during compilation.

This option is applied if compiling using PICC, the command-line driver and the `--MSGDISABLE` driver option, see **Section 2.7.37 "--MSGDISABLE: Disable Warning Messages"**.

See **Section 2.6 "Compiler Messages"** for full information about the compiler's messaging system.

5.2.28   **--EDF=***message file***: Set Message File Path**

This option is mainly used by the command-line driver, PICC, to specify the path of the message description file. The default file is located in the `dat` directory in the compiler's installation directory.

See **Section 2.6 "Compiler Messages"** for full information about the compiler's messaging system.

5.2.29 --EMAX=*number*: Specify Maximum Number of Errors

This option is mainly used by the command-line driver, PICC, to specify the maximum number of errors that can be encountered before the assembler terminates. The default number is 10 errors.

This option is applied if compiling using PICC, the command-line driver and the `--ERRORS` driver option, see **Section 2.7.28 "--ERRORS: Maximum Number of Errors"**.

See **Section 2.6 "Compiler Messages"** for full information about the compiler's messaging system.

5.2.30 --NORLF: Do Not Relocate List File

Use of this option prevents the linker applying fixups to the assembly list file produced by the assembler. This option is normally using by the command line driver, PICC, when performing pre-link stages, but is omitted when performing the final link step so that the list file shows the final absolute addresses.

If you are attempting to resolve fixup errors, this option should be disabled so as to fixup the assembly list file and allow absolute addresses to be calculated for this file. If the compiler driver detects the presence of a preprocessor macro `__DEBUG` which is equated to 1, then this option will be disabled when building. This macro is set when choosing a *Debug* build in MPLAB IDE, so always have this selected if you encounter such errors.

5.2.31 --VER: Print Version Number

This option printed information relating to the version and build of the linker. The linker will terminate after processing this option, even if other options and files are present on the command line.

5.3 RELOCATION AND PSECTS

This section looks at the input files that the linker has to work with.

The linker can read both relocatable object files and object-file libraries (`.lib` extension). The library files are a collection of object files packaged into a single unit, so essentially we only need consider the format of object files.

Each object file consists of a number of records. Each record has a type that indicates what sort of information it holds. Some record types hold general information about the target device and its configuration, other records types may hold data, and others, program debugging information, for example.

A lot of the information in object files relates to psects. Psects are an assembly domain construct and are essentially a block of something, either instructions or data. Everything that contributes to the program is located in a psect. See **Section 4.3.8 "Program Sections"**. There is a particular record type that is used to hold the data in psects. The bulk of each object file consists of psect records containing the executable code and variables etc.

We are now in a position to look at the fundamental tasks the linker performs, which are:

• combining all the relocatable object files into one
• relocation of psects contained in the object files into memory
• fixup of symbolic references in the psects

There is typically at least two object files that are passed to the linker. One will be produced from all the C code in the project, including C library code. There is only one of these files since the code generator compiles and combines all the C code of the program and produces just the one assembly output. The other file passed to the linker will be the object code produced from the runtime startup code, see **Section 2.4.2 "Runtime Startup Code"**.

If there are assembly source files in the project, then there will also be one object file produced for each source file and these will be passed to the linker. Existing object files, or object file libraries can also be specified in a project, and if present, these will also be passed to the linker.

The output of the linker is also an object file, but there is only ever one file produced. The file is absolute since relocation will have been performed by the linker. The output file will consist of the information from all input object files merged together.

Relocation consists of placing the psect data into the memory of the target device.

The target device memory specification is passed to the linker by the way of linker options. These options are generated by the command-line driver, PICC. There are no linker scripts or means of specifying options in any source file. The default linker options rarely need adjusting, but can be changed, if required and with caution, using the driver option `-L-`, see **Section 2.7.7 "-L-: Adjust Linker Options Directly"**.

Once psects are placed at actual memory locations, symbolic references made in the psects data can be replaced with absolute values. This is a process called fixup.

For each psect record in the object file, there is a corresponding relocation record that indicates which bytes (or bits) in the psect record need to be adjusted once relocation is complete. The relocation records also specify how the values are to be determined. A linker fixup overflow error can occur if the value determined by the linker is too large to fit in the "hole" reserved for the value in the psect. See **Section "(477) fixup overflow in expression (location 0x* (0x*+*), size *, value 0x*) (Linker)"** for information on finding the cause of these errors.

5.4 MAP FILES

The map file contains information relating to the relocation of psects and the addresses assigned to symbols within those psects.

5.4.1 Generation

If compilation is being performed via an IDE such as HI-TIDE or MPLAB IDE, a map file is generated by default without you having to adjust the compiler options. If you are using the driver from the command line then you'll need to use the `-M` option to request that the map file be produced, see **Section 5.2.15 "-Mmapfile"**. Map files use the extension `.map`.

Map files are produced by the linker. If the compilation process is stopped before the linker is executed, then no map file is produced. The linker will still produce a map file even if it encounters errors, which will allow you to use this file to track down the cause of the errors. However, if the linker ultimately reports `too many errors` then it did not run to completion, and the map file will be either not created or not complete. You can use the `--ERRORS` option (see **Section 2.7.28 "--ERRORS: Maximum Number of Errors"**) on the command line to increase the number of errors before the linker exits.

5.4.2   Contents

The sections in the map file, in order of appearance, are as follows.

* The compiler name and version number
* A copy of the command line used to invoke the linker
* The version number of the object code in the first file linked
* The machine type
* The call graph information
* A psect summary sorted by the psect's parent object file
* A psect summary sorted by the psect's CLASS
* A segment summary
* Unused address ranges summary
* The symbol table

Portions of an example map file, along with explanatory text, are shown in the following sections.

5.4.2.1   GENERAL INFORMATION

At the top of the map file is general information relating to the execution of the linker.

When analyzing a program, always confirm the compiler version number shown in the map file if you have more than one compiler version installed to ensure the desired compiler is being executed.

The device selected with the `--CHIP` option (**Section 2.7.20 "--CHIP: Define Processor"**) or that select in your IDE, should appear after the Machine type entry.

The *object code version* relates to the file format used by relocatable object files produced by the assembler. Unless either the assembler or linker have been updated independently, this should not be of concern.

A typical map file may begin something like the following. This example has been cut down for clarity.

```
--edf=C:\Program Files\HI-TECH Software\PICC\PRO\9.65pl1\dat\en_msgs.txt \
-cs -h+test.sym -z -Q16F946 -ol.obj -Mtest.map -E1 -ACODE=00h-07FFhx4 \
-ACONST=00h-0FFhx32 -AENTRY=00h-0FFhx32 -ASTRING=00h-0FFhx32 \
-ARAM=020h-06Fh,0A0h-0EFh,0120h-016Fh,01A0h-01EFh \
-AABS1=020h-07Fh,0A0h-0EFh,0120h-016Fh,01A0h-01EFh -ABANK0=020h-07Fh \
-ABANK1=0A0h-0EFh -ABANK2=0120h-016Fh -ABANK3=01A0h-01EFh \
-ACOMMON=070h-07Fh \
-preset_vec=00h,intentry,intcode,intret,init,init23,end_init,...\
-pstrings=CODE -ppowerup=CODE -ptemp=-COMMON -pcommon=-COMMON \
-prbss_0=BANK0,rbit_0=BANK0,rdata_0=BANK0,idata_0=CODE -pnvram=BANK0 \
-prbss_1=BANK1,rbit_1=BANK1,rdata_1=BANK1,idata_1=CODE \
-pnvram_1=BANK1,nvbit_1=BANK1 \
-ACONFIG=02007h-02007h -pconfig=CONFIG -DCONFIG=2 -AIDLOC=02000h-02003h \
-pidloc=IDLOC -DIDLOC=2 -AEEDATA=00h-0FFh/02100h -peeprom_data=EEDATA \
-DEEDATA=2 -pfloat_text0=CODE,float_text1=CODE,float_text2=CODE \
-pfloat_text3=CODE,float_text4=CODE -DCODE=2 startup.obj test.obj

Object code version is 3.10

Machine type is 16F946
```

The *Linker command line* shows all the command-line options and files that were passed to the linker for the last build. Remember, these are linker options and not command-line driver options.

The linker options are necessarily complex. Fortunately, they rarely need adjusting from their default settings. They are formed by the command-line driver, PICC, based on the selected target device and the specified driver options. You can often confirm that driver options were valid by looking at the linker options in the map file. For example, if you ask the driver to reserve an area of memory, you should see a change in the linker options used.

If the default linker options must be changed, this can be done indirectly through the driver using the driver `-L-` option, see **Section 2.7.7 "-L-: Adjust Linker Options Directly"**. If you use this option, always confirm the change appears correctly in the map file.

5.4.2.2   PSECT INFORMATION LISTED BY MODULE

The next section in the map file lists those modules that made a contribution to the output, and information regarding the psects these modules defined.

This section is heralded by the line that contains the headings:

```
Name   Link  Load  Length  Selector  Space  Scale
```

Under this on the far left is a list of object files. These object files include both files generated from source modules and those that were extracted from object library files (`.lib` extension). In the latter case, the name of the library file is printed before the object file list. Note that since the code generator combines all C source files (and p-code libraries), there will only be one object file representing the entire C part of the program. The object file corresponding to the runtime startup code is normally present in this list.

The information in this section of the map file can be used to confirm that a module is making a contribution to the output file and to determine the exact psects that each module defines.

Shown are all the psects (under the *Name* column) that were linked into the program from each object file, and information about that psect.

The linker deals with two kinds of addresses: link and load. Generally speaking the link address of a psect is the address by which it will be accessed at run time.

The load address, which is often the same as the link address, is the address at which the psect will start within the output file (HEX or binary file etc.). If a psect is used to hold bits, the load address is irrelevant and is instead used to hold the link address (in bit units) converted into a byte address.

The *Length* of the psect is shown in the units used by that psect.

The *Selector* is less commonly used and is of no concern when compiling for PIC devices.

The *Space* field is important as it indicates the memory space in which the psect was placed. For Harvard architecture machines, with separate memory spaces (such as the PIC10/12/16 devices), this field must be used in conjunction with the address to specify an exact storage location. A space of 0 indicates the program memory, and a space of 1 indicates the data memory. See **Section 4.3.9.3.12 "Space"**.

The *Scale* of a psect indicates the number of address units per byte. This is left blank if the scale is 1 and will show 8 for psects that hold bit objects. The load address of psects that hold bits is used to display the link address converted into units of bytes, rather than the load address. See **Section 4.3.9.3.2 "Bit"**.

For example, the following appears in a map file.

```
        Name  Link  Load  Length Selector  Space  Scale
ext.obj text   3A    3A      22       30      0
        bss    4B    4B      10       4B      1
        rbit   50     A       2        0      1      8
```

This indicates that one of the files that the linker processed was called `ext.obj`. (This may have been derived from C code or a source file called `ext.as`.)

This object file contained a `text` psect, as well as psects called `bss` and `rbit`.

The psect `text` was linked at address 3A and `bss` at address 4B. At first glance, this seems to be a problem given that text is 22 words long, however note that they are in different memory areas, as indicated by the space flag (0 for `text` and 1 for `bss`), and so do not occupy the same memory.

The psect `rbit` contains bit objects, and this can be confirmed by looking it the scale value, which is 8. Again, at first glance there seems there could be an issue with `rbit` linked over the top of `bss`. Their space flags are the same, but since `rbit` contains bit objects, its link address is in units of bits. Note that the load address field of `rbit` psect displays the link address converted to byte units, i.e. 50h/8 => Ah.

Underneath the object file list there may be a label *COMMON*. This shows the contribution to the program from program-wide psects, in particular that used by the compiled stack.

5.4.2.3   PSECT INFORMATION LISTED BY CLASS

The next section in the map file shows the same psect information but grouped by the psects' class.

This section is heralded by the line that contains the headings:

```
TOTAL   Name  Link  Load  Length
```

Under this are the class names followed by those psects which belong to this class, see **Section 4.3.9.3.3 "Class"**. These psects are the same as those listed by module in the above section; there is no new information contained in this section, just a different presentation.

5.4.2.4   SEGMENT LISTING

The class listing in the map file is followed by a listing of segments. A segment is conceptual grouping of contiguous psects in the same memory space, and are used by the linker as an aid in psect placement. There is no segment assembler directive and segments cannot be controlled in any way.

This section is heralded by the line that contains the headings:

```
SEGMENTS   Name  Load  Length  Top  Selector  Space  Class
```

The name of a segment is derived from the psect in the contiguous group with the lowest link address. This can lead to confusion with the psect with the same name. Do not read psect information from this section of the map file.

Typically this section of the map file can be ignored by the user.

5.4.2.5   UNUSED ADDRESS RANGES

The last of the memory summaries show the memory is has not been allocated, and is hence unused. The linker is aware of any memory allocated by the code generator (for absolute variables), and so this free space is accurate.

This section follows the heading:

```
UNUSED ADDRESS RANGES
```

and is followed by a list of classes and the memory still available in each class. If there is more than one memory range available in a class, each range is printed on a separate line. Any paging boundaries within a class are not displayed, but the column *Largest block* shows the largest contiguous free space which takes into account any paging in the memory range. If you are looking to see why psects cannot be placed into memory (e.g. cant-find-space type errors) then this important information to study.

Note that the memory associated with a class can overlap that in others, thus the total free space is not simply the addition of all the unused ranges.

5.4.2.6   SYMBOL TABLE

The final section in the map file list global symbols that the program defines. This section has a heading:

```
Symbol Table
```

and is followed by two columns in which the symbols are alphabetically listed. As always with the linker, any C derived symbol is shown with its assembler equivalent symbol name. See **Section 3.9.3.1 "Equivalent Assembly Symbols"**.

The symbols listed in this table are:

- Global assembly labels
- Global EQU /SET assembler directive labels
- Linker-defined symbols

Assembly symbols are made global via the GLOBAL assembler directive, see **Section 4.3.9.1 "GLOBAL"** for more information.

Linker-defined symbols act like EQU directives, however they are defined by the linker during the link process, and no definition for them will appear in any source or intermediate file. See **Section 3.11.3 "Linker-Defined Symbols"**.

Each symbol is shown with the psect in which they are placed, and the value (usually an address) which the symbol has been assigned. There is no information encoded into a symbol to indicate whether it represents code or data, nor in which memory space it resides.

If the psect of a symbol is shown as (abs), this implies that the symbol is not directly associated with a psect. Such is the case for absolute C variables, or any symbols that are defined using an EQU directive in assembly.

Note that a symbol table is also shown in each assembler list file. (See **Section 2.7.17 "--ASMLIST: Generate Assembler List Files"** for information on generating these files.) These differ to that shown in the map file in that they list also list local symbols, and only show symbols defined in the corresponding module.

NOTES:

Chapter 6. Utilities

6.1 INTRODUCTION

This chapters discusses some of the utility applications that are bundled with the compiler.

Some of these applications may not be normally invoked when building, but can be manually executed to perform certain tasks.

6.2 LIBRARIAN

The librarian program, `LIBR`, has the function of combining several files into a single file known as a library. The reasons you might want to use a library in a project are:

• there will be fewer files to link
• the file content will be accessed faster
• libraries uses less disk space

The librarian can build p-code libraries (`.lpp` extension) from p-code files (`.p1` extension), or object code libraries (`.lib` extension) from object files (`.obj` extension). P-code libraries should be only created if all the library source code is written in C. Object code libraries should be used for assembly code that is to be built into a library.

With both library types, only those modules required by a program will be extracted and included in the program output.

6.2.1 The Library Format

The modules in a library are simply concatenated, but a directory of the modules and symbols in the library is maintained at the beginning of a library file. Since this directory is smaller than the sum of the modules, on the first pass the linker can perform faster searches just reading the directory, and not all the modules. On the second pass it need read only those modules which are required, seeking over the others. This all minimizes disk I/O when linking.

It should be noted that the library format is not a general purpose archiving mechanism as is used by some other compiler systems. This has the advantage that the format may be optimized toward speeding up the linkage process.

6.2.2 Using the Librarian

Library files can be built directly using the command-line driver, see **Section 2.7.44 "--OUTPUT= type: Specify Output File Type"**. In this case the driver will invoke `LIBR` with the appropriate options saving you from having to use the librarian directly. You may wish to perform this step manually, or you may need to look at the contents of library files, for example. This section shows how the librarian can be executed from the command-line. The librarian cannot be called from IDEs, such as MPLAB IDE.

The librarian program is called `LIBR`, and the formats of commands to it are as follows:

```
LIBR [options]  k  file.lpp [file1.p1 file2.p1...]
LIBR [options]  k  file.lib [file1.obj file2.obj ...]
```

The *options* are zero or more librarian options which affect the output of the program. These are listed in Table 6-1.

TABLE 6-1:   LIBRARIAN COMMAND-LINE OPTIONS

Option	Effect
-P *width*	Specify page width
-W	Suppress non-fatal errors

A key letter, `k`, denotes the command requested of the librarian (replacing, extracting or deleting modules, listing modules or symbols). These commands are listed in Table 6-2.

TABLE 6-2:   LIBRARIAN KEY LETTER COMMANDS

Key	Meaning
r	Replace modules
d	Delete modules
x	Extract modules
m	List modules
s	List modules with symbols
o	Re-order modules

The first file name listed after the key is the name of the library file to be used. The following files, if required, are the modules of the library required by the command specified.

If you are building a p-code library, the modules listed must be p-code files. If you are building an object file library, the modules listed must be object files.

When replacing or extracting modules, the names of the modules to be replaced or extracted must be specified. If no names are supplied, all the modules in the library will be replaced or extracted respectively.

Adding a file to a library is performed by requesting the librarian to replace it in the library. Since it is not present, the module will be appended to the library. If the `r` key is used and the library does not exist, it will be created.

When using the `d` key letter, the named modules will be deleted from the library. In this instance, it is an error not to give any module names.

The `m` and `s` key letters will list the named modules and, in the case of the `s` key letter, the global symbols defined or referenced within. A `D` or `U` letter is used to indicate whether each symbol is defined in the module, or referenced but undefined. As with the `r` and `x` key letters, an empty list of modules means all the modules in the library.

The `o` key takes a list of module names and re-orders the matching modules in the library file so they have the same order as that listed on the command line. Modules which are not listed are left in their existing order, and will appear after the re-ordered modules.

6.2.2.1 EXAMPLES

Here are some examples of usage of the librarian. The following command:

```
LIBR s htpic--c.lpp ctime.p1
```

lists the global symbols in the modules `ctime.p1`, as shown here:

```
ctime.p1                          D _moninit
                                  D _localtime
                                  D _gmtime
                                  D _asctime
                                  D _ctime
```

The `D` letter before each symbol indicates that these symbols are defined by the module.

Using the command above without specifying the module name will list all the symbols defined (or undefined) in the library.

The following command deletes the object modules `a.obj`, `b.obj` and `c.obj` from the library `lcd.lib`:

```
LIBR d lcd.lib a.obj b.obj c.obj
```

6.2.3 Supplying Arguments

Since it is often necessary to supply many object file arguments to `LIBR`, arguments will be read from standard input if no command-line arguments are given. If the standard input is attached to the console, `LIBR` will prompt for input.

Multiple line input may be given by using a *backslash* as a continuation character on the end of a line. If standard input is redirected from a file, `LIBR` will take input from the file, without prompting. For example:

```
libr
libr> r file.lib 1.obj 2.obj 3.obj \
libr> 4.obj 5.obj 6.obj
```

will perform much the same as if the object files had been typed on the command line. The `libr>` prompts were printed by `LIBR` itself, the remainder of the text was typed as input.

```
libr <lib.cmd
```

`LIBR` will read input from `lib.cmd`, and execute the command found therein. This allows a virtually unlimited length command to be given to `LIBR`.

6.2.4 Ordering of Libraries

The librarian creates libraries with the modules in the order in which they were given on the command line. When updating a library the order of the modules is preserved. Any new modules added to a library after it has been created will be appended to the end.

The ordering of the modules in a library is significant to the linker. If a library contains a module which references a symbol defined in another module in the same library, the module defining the symbol should come after the module referencing the symbol.

6.2.5 Error Messages

`LIBR` issues various error messages, most of which represent a fatal error, while some represent a harmless occurrence which will nonetheless be reported unless the `-W` option was used. In this case all warning messages will be suppressed.

6.3   OBJTOHEX

The HI-TECH linker is capable of producing object files as output. Any other format
required must be produced by running the utility program OBJTOHEX. This allows con-
version of object files as produced by the linker into a variety of different formats,
including various HEX formats. The program is invoked thus:

```
OBJTOHEX [options]  inputfile  outputfile
```

All of the arguments are optional. The options for OBJTOHEX are listed in Table 6-3.

TABLE 6-3:   OBJTOHEX COMMAND-LINE OPTIONS

Option	Meaning
-8	Produce a CP/M-86 output file
-A	Produce an ATDOS .atx output file
-B*base*	Produce a binary file with offset of *base*. Default file name is l.obj
-C*ckfile*	Read a list of checksum specifications from *ckfile* or standard input
-D	Produce a COD file
-E	Produce an MS-DOS .exe file
-F*fill*	Fill unused memory with words of value *fill* - default value is 0FFh
-I	Produce an *Intel* HEX file with linear addressed extended records.
-L	Pass relocation information into the output file (used with .exe files)
-M	Produce a *Motorola* HEX file (S19, S28 or S37 format)
-N	Produce an output file for Minix
-P*stk*	Produce an output file for an *Atari* ST, with optional stack size
-R	Include relocation information in the output file
-S*file*	Write a symbol file into *file*
-T	Produce a *Tektronix* HEX file.
-TE	Produce an extended TekHEX file.
-U	Produce a COFF output file
-UB	Produce a UBROF format file
-V	Reverse the order of words and long words in the output file
-*n,m*	Format either Motorola or Intel HEX file, where *n* is the maximum number of bytes per record and *m* specifies the record size rounding. Non-rounded records are zero padded to a multiple of *m*. *m* itself must be a multiple of 2.
--EDF	Specify message file location
--EMAX	Specify maximum number of errors
--MSGDISABLE	Specify disabled messages
--VER	Print version number and stop

If *outputfile* is omitted it defaults to l.HEX or l.bin depending on whether the -b
option is used. The *inputfile* defaults to l.obj.

Except where noted, any address will be interpreted as a decimal value. To force inter-
pretation as a HEX number, a trailing H, or h, should be added, e.g. 765FH will be
treated as a HEX number.

6.3.1 Checksum Specifications

If you are generating a HEX file output, use HEXMATE's checksum tools, described in **Section 6.6 "HEXMATE"**.

For other file formats, the OBJTOHEX checksum specification allows automated checksum calculation and takes the form of several lines, each line describing one checksum. The syntax of a checksum line is:

```
addr1-addr2 where1-where2 +offset
```

All of *addr1*, *addr2*, *where1*, *where2* and *offset* are HEX numbers, without the usual H suffix.

Such a specification says that the bytes at *addr1* through to *addr2* inclusive should be summed and the sum placed in the locations *where1* through *where2* inclusive. For an 8 bit checksum these two addresses should be the same. For a checksum stored low byte first, *where1* should be less than *where2*, and vice versa.

The *+offset* value is optional, but if supplied, the value will be used to initialize the checksum. Otherwise it is initialized to zero.

For example:

```
0005-1FFF 3-4 +1FFF
```

This will sum the bytes in 5 through 1FFFH inclusive, then add 1FFFH to the sum. The 16 bit checksum will be placed in locations 3 and 4, low byte in 3. The checksum is initialized with 1FFFH to provide protection against an all zero ROM, or a ROM misplaced in memory. A run time check of this checksum would add the last address of the ROM being checksummed into the checksum. For the ROM in question, this should be 1FFFH. The initialization value may, however, be used in any desired fashion.

6.4 CREF

The cross reference list utility, CREF, is used to format raw cross-reference information produced by the compiler or the assembler into a sorted listing.

A raw cross-reference file is produced with the --CR command-line driver option. The assembler will generate a raw cross-reference file with a -C assembler option or a XREF control line.

The general form of the CREF command is:

```
cref [options]  files
```

where *options* is zero or more options as described below and *files* is one or more raw cross-reference files.

CREF will accept wildcard filenames and I/O redirection. Long command lines may be supplied by invoking CREF with no arguments and typing the command line in response to the cref> prompt. A *backslash* at the end of the line will be interpreted to mean that more command lines follow.

CREF takes the options listed in **Section Table 6-4: "CREF command-line options"**.

TABLE 6-4:   CREF COMMAND-LINE OPTIONS

Option	Meaning
–F*prefix*	Exclude symbols from files with a pathname or filename starting with *prefix*
–H*heading*	Specify a heading for the listing file
–L*len*	Specify the page length for the listing file
–O*outfile*	Specify the name of the listing file
–P*width*	Set the listing width
–S*stoplist*	Read file *stoplist* and ignore any symbols listed.
–X*prefix*	Exclude any symbols starting with *prefix*
--EDF	Specify message file location
--EMAX	Specify maximum number of errors
--MSGDISABLE	Specify disabled messages
--VER	Print version number and stop

Each option is described in more detail in the following sections.

6.4.1   -F*prefix*

It is often desired to exclude from the cross-reference listing any symbols defined in a system header file, e.g. <stdio.h>. The -F option allows specification of a path name prefix that will be used to exclude any symbols defined in a file whose path name begins with that prefix. For example, -F\ will exclude any symbols from all files with a path name starting with \.

6.4.2   -H*heading*

The -H option takes a string as an argument which will be used as a header in the listing. The default heading is the name of the first raw cross-ref information file specified.

6.4.3   -L*len*

Specify the length of the paper on which the listing is to be produced, e.g. if the listing is to be printed on 55 line paper you would use a -L55 option. The default is 66 lines.

6.4.4   -O*outfile*

Allows specification of the output file name. By default the listing will be written to the standard output and may be redirected in the usual manner. Alternatively *outfile* may be specified as the output file name.

6.4.5   -P*width*

This option allows the specification of the width to which the listing is to be formatted, e.g. -P132 will format the listing for a 132 column printer. The default is 80 columns.

6.4.6   -S*stoplist*

The -S option should have as its argument the name of a file containing a list of symbols not to be listed in the cross-reference. Symbols should be listed, one per line in the file. Use the C domain symbols. Multiple stoplists may be supplied with multiple -S options.

6.4.7   -X*prefix*

The -X option allows the exclusion of symbols from the listing, based on a prefix given as argument to -X. For example if it was desired to exclude all symbols starting with the character sequence xyz then the option -Xxyz would be used. If a digit appears in the character sequence then this will match any digit in the symbol, e.g. -XX0 would exclude any symbols starting with the letter x followed by a digit.

6.4.8   --EDF=*message file*: Set Message File Path

This option is mainly used by the command-line driver, PICC, to specify the path of the message description file. The default file is located in the dat directory in the compiler's installation directory.

See **Section 2.6 "Compiler Messages"** for full information about the compiler's messaging system.

6.4.9   --EMAX=*number*: Specify Maximum Number of Errors

This option is mainly used by the command-line driver, PICC, to specify the maximum number of errors that can be encountered before CREF terminates. The default number is 10 errors.

This option is applied if compiling using PICC, the command-line driver and the --ERRORS driver option, see **Section 2.7.28 "--ERRORS: Maximum Number of Errors"**.

See **Section 2.6 "Compiler Messages"** for full information about the compiler's messaging system.

6.4.10   --MSGDISABLE=*message numbers* Disable Messages

This option is mainly used by the command-line driver, PICC, to disable particular message numbers. It takes a *comma*-separate list of message numbers that will be disabled during compilation.

This option is applied if compiling using PICC, the command-line driver and the --MSGDISABLE driver option, see **Section 2.7.37 "--MSGDISABLE: Disable Warning Messages"**.

See **Section 2.6 "Compiler Messages"** for full information about the compiler's messaging system.

6.4.11   --VER: Print Version Number

This option prints information relating to the version and build of CREF. CREF will terminate after processing this option, even if other options and files are present on the command line.

6.5 CROMWELL

The CROMWELL utility converts code and symbol files into different formats. These files are typically used by debuggers and allow source-level debugging of code. The output formats available are shown in Table 6-5.

TABLE 6-5: CROMWELL FORMAT TYPES

Key	Format
cod	*Bytecraft* COD file
coff	COFF file format
elf	ELF/DWARF file
eomf51	Extended OMF-51 format
hitech	HI-TECH Software format
icoff	ICOFF file format
ihex	*Intel* HEX file format
mcoff	Microchip COFF file format
omf51	OMF-51 file format
pe	P&E file format
s19	*Motorola* HEX file format

The CROMWELL application is automatically executed by the command-line driver when required. The following information is required if running the application manually.

The general form of the CROMWELL command is:

CROMWELL [*options*] *inputFiles* -o*key* [*outputFile*]

where *options* can be any of the options shown in Table 6-6.

TABLE 6-6: CROMWELL COMMAND-LINE OPTIONS

Option	Description
-P*name[,architecture]*	Processor name and architecture
-N	Identify code classes
-D	Dump input file
-C	Identify input files only
-F	Fake local symbols as global
-O*key*	Set the output format
-I*key*	Set the input format
-L	List the available formats
-E	Strip file extensions
-B	Specify big-endian byte ordering
-M	Strip underscore character
-V	Verbose mode
--EDF=*path*	Specify message file location
--EMAX=*number*	Specify maximum number of errors
--MSGDISABLE=*list*	Specify disabled messages
--VER	Print version number and stop

The *outputFile* (optional) is the name of the output file. The *inputFiles* are typically the HEX and SYM file.

CROMWELL automatically searches for the SDB files and reads those if they are found. The options are further described in the following paragraphs.

6.5.1   -Pname[,architecture]

The -P options takes a string which is the name of the processor used. CROMWELL may use this in the generation of the output format selected.

Note that to produce output in COFF format an additional argument to this option which also specifies the processor architecture is required. Hence for this format the usage of this option must take the form: *-Pname,architecture*. Table 6-7 enumerates the architectures supported for producing COFF files.

TABLE 6-7:   ARCHITECTURE ARGUMENTS

Architecture	Description
PIC12	Microchip baseline PIC® MCU chips
PIC14	Microchip Mid-Range PIC MCU chips
PIC14E	Microchip Enhanced Mid-Range PIC MCU chips
PIC16	Microchip high-end (17CXXX) PIC MCU chips
PIC18	Microchip PIC18 chips
PIC24	Microchip PIC24F and PIC24H chips
PIC30	Microchip dsPIC30 and dsPIC33 chips

6.5.2   -N

To produce some output file formats (e.g. COFF), CROMWELL requires that the names of the program memory space psect classes be provided. The names of the classes are specified as a *comma*-separated list. See the map file (**Section 5.4 "Map Files"**) to determine which classes the linker uses.

For example, mid-range devices typically requires -NCODE,CONST,ENTRY,STRING.

6.5.3   -D

The -D option is used to display details about the named input file in a human-readable format.This option is useful if you need to check the contents of the file, which are usually binary files. The input file can be one of the file types as shown in Table 6-5.

6.5.4   -C

This option will attempt to identify if the specified input files are one of the formats as shown in Table 6-5. If the file is recognized, a confirmation of its type will be displayed.

6.5.5   -F

When generating a COD file, this option can be used to force all local symbols to be represented as global symbols. The may be useful where an emulator cannot read local symbol information from the COD file.

6.5.6   -Okey

This option specifies the format of the output file. The *key* can be any of the types listed in Table 6-5.

6.5.7   -Ikey

This option can be used to specify the default input file format. The *key* can be any of the types listed in Table 6-5.

6.5.8 -L

Use this option to show what file format types are supported. A list similar to that given in Table 6-5 will be shown.

6.5.9 -E

Use this option to tell CROMWELL to ignore any filename extensions that were given. The default extension will be used instead.

6.5.10 -B

In formats that support different endian types, use this option to specify big-endian byte ordering.

6.5.11 -M

When generating COD files this option will remove the preceding underscore character from symbols.

6.5.12 -V

Turns on verbose mode which will display information about operations CROMWELL is performing.

6.5.13 --EDF=*message file*: Set Message File Path

This option is mainly used by the command-line driver, PICC, to specify the path of the message description file. The default file is located in the dat directory in the compiler's installation directory.

See **Section 2.6 "Compiler Messages"** for full information about the compiler's messaging system.

6.5.14 --EMAX=*number*: Specify Maximum Number of Errors

This option is mainly used by the command-line driver, PICC, to specify the maximum number of errors that can be encountered before CROMWELL terminates. The default number is 10 errors.

This option is applied if compiling using PICC, the command-line driver and the --ERRORS driver option, see **Section 2.7.28 "--ERRORS: Maximum Number of Errors"**.

See **Section 2.6 "Compiler Messages"** for full information about the compiler's messaging system.

6.5.15 --MSGDISABLE=*message numbers* Disable Messages

This option is mainly used by the command-line driver, PICC, to disable particular message numbers. It takes a *comma*-separate list of message numbers that will be disabled during compilation.

This option is applied if compiling using PICC, the command-line driver and the --MSGDISABLE driver option, see **Section 2.7.37 "--MSGDISABLE: Disable Warning Messages"**.

See **Section 2.6 "Compiler Messages"** for full information about the compiler's messaging system.

6.5.16 --VER: Print Version Number

This option printed information relating to the version and build of CROMWELL. CROMWELL will terminate after processing this option, even if other options and files are present on the command line.

6.6 HEXMATE

The HEXMATE utility is a program designed to manipulate Intel HEX files. HEXMATE is a post-link stage utility which is automatically invoked by the compiler driver, and that provides the facility to:

- Calculate and store variable-length checksum values
- Fill unused memory locations with known data sequences
- Merge multiple Intel HEX files into one output file
- Convert INHX32 files to other INHX formats (e.g. INHX8M)
- Detect specific or partial opcode sequences within a HEX file
- Find/replace specific or partial opcode sequences
- Provide a map of addresses used in a HEX file
- Change or fix the length of data records in a HEX file.
- Validate checksums within Intel HEX files.

Typical applications for HEXMATE might include:

- Merging a bootloader or debug module into a main application at build time
- Calculating a checksum over a range of program memory and storing its value in program memory or EEPROM
- Filling unused memory locations with an instruction to send the PC to a known location if it gets lost.
- Storage of a serial number at a fixed address.
- Storage of a string (e.g. time stamp) at a fixed address.
- Store initial values at a particular memory address (e.g. initialize EEPROM)
- Detecting usage of a buggy/restricted instruction
- Adjusting HEX file to meet requirements of particular bootloaders

6.6.1 HEXMATE Command Line Options

HEXMATE is automatically called by the command line driver, PICC. This is primarily to merge in HEX files with the output generated by the source files, however there are some PICC options which directly map to HEXMATE options, and so other functionality can be requested without having to run HEXMATE explicitly on the command line. For other functionality, the following details the options available when running this application.

If HEXMATE is to be run directly, its usage is:

```
HEXMATE [specs,]file1.HEX [[specs,]file2.HEX ... [specs,]fileN.HEX]
[options]
```

Where *file1.HEX* through to *fileN.HEX* form a list of input Intel HEX files to merge using HEXMATE. If only one HEX file is specified, then no merging takes place, but other functionality is specified by additional options. Table 6-8 lists the command line options that HEXMATE accepts.

TABLE 6-8: HEXMATE COMMAND-LINE OPTIONS

Option	Effect
-ADDRESSING	Set address fields in all HEXMATE options to use word addressing or other
-BREAK	Break continuous data so that a new record begins at a set address
-CK	Calculate and store a checksum value
-FILL	Program unused locations with a known value
-FIND	Search and notify if a particular code sequence is detected

TABLE 6-8: HEXMATE **COMMAND-LINE OPTIONS (CONTINUED)**

Option	Effect
-FIND...,DELETE	Remove the code sequence if it is detected (use with caution)
-FIND...,REPLACE	Replace the code sequence with a new code sequence
-FORMAT	Specify maximum data record length or select INHX variant
-HELP	Show all options or display help message for specific option
-LOGFILE	Save HEXMATE analysis of output and various results to a file
-O*file*	Specify the name of the output file
-SERIAL	Store a serial number or code sequence at a fixed address
-SIZE	Report the number of bytes of data contained in the resultant HEX image.
-STRING	Store an ASCII string at a fixed address
-STRPACK	Store an ASCII string at a fixed address using string packing
-W	Adjust warning sensitivity
+	Prefix to any option to overwrite other data in its address range if necessary

The input parameters to HEXMATE are now discussed in greater detail. Note that any integral values supplied to the HEXMATE options should be entered as hexadecimal values without leading 0x or trailing h characters. Note also that any address fields specified in these options are to be entered as byte addresses, unless specified otherwise in the -ADDRESSING option.

6.6.1.1   SPECIFICATIONS,FILENAME.HEX

Intel HEX files that can be processed by HEXMATE should be in either INHX32 or INHX8M format. Additional specifications can be applied to each HEX file to put restrictions or conditions on how this file should be processed.

If any specifications are used they must precede the filename. The list of specifications will then be separated from the filename by a *comma*.

A *range restriction* can be applied with the specification r*Start-End*. A range restriction will cause only the address data falling within this range to be used. For example:

```
r100-1FF,myfile.hex
```

will use myfile.hex as input, but only process data which is addressed within the range 100h-1FFh (inclusive) from that file.

An address shift can be applied with the specification s*Offset*. If an address shift is used, data read from this HEX file will be shifted (by the offset specified) to a new address when generating the output. The offset can be either positive or negative. For example:

```
r100-1FFs2000,myfile.HEX
```

will shift the block of data from 100h-1FFh to the new address range 2100h-21FFh.

Be careful when shifting sections of executable code. Program code should only be shifted if it is position independent.

6.6.1.2   + PREFIX

When the + operator precedes an argument or input file, the data obtained from that source will be forced into the output file and will overwrite another other data existing at that address range. For example:

```
+input.HEX +-STRING@1000="My string"
```

Ordinarily, HEXMATE will issue an error if two sources try to store differing data at the same location. Using the + operator informs HEXMATE that if more than one data source tries to store data to the same address, the one specified with a + prefix will take priority.

6.6.1.3   -ADDRESSING

By default, all address arguments in HEXMATE options expect that values will be entered as byte addresses. In some device architectures the native addressing format may be something other than byte addressing. In these cases it would be much simpler to be able to enter address-components in the device's native format. To facilitate this, the -ADDRESSING option is used.

This option takes exactly one parameter which configures the number of bytes contained per address location. If for example a device's program memory naturally used a 16-bit (2 byte) word-addressing format, the option -ADDRESSING=2 will configure HEXMATE to interpret all command line address fields as word addresses. The affect of this setting is global and all HEXMATE options will now interpret addresses according to this setting. This option will allow specification of addressing modes from one byte per address to four bytes per address.

6.6.1.4   -BREAK

This option takes a *comma*-separated list of addresses. If any of these addresses are encountered in the HEX file, the current data record will conclude and a new data record will recommence from the nominated address. This can be useful to use new data records to force a distinction between functionally different areas of program space. Some HEX file readers depend on this.

6.6.1.5   -CK

The -CK option is for calculating a checksum. The usage of this option is:

```
-CK=start-end@destination [+offset][wWidth][tCode][gAlogithm]
```

where:

- *start* and *end* specify the address range over which the checksum will be calculated.
- *destination* is the address where the checksum result will be stored. This value cannot be within the range of calculation.
- *offset* is an optional initial value to add to the checksum result.
- *Width* is optional and specifies the byte-width of the checksum result. Results can be calculated for byte-widths of 1 to 4 bytes. If a positive width is requested, the result will be stored in big-endian byte order. A negative width will cause the result to be stored in little-endian byte order. If the width is left unspecified, the result will be 2 bytes wide and stored in little-endian byte order.
- *Code* is a hexadecimal code that will trail each byte in the checksum result. This can allow each byte of the checksum result to be embedded within an instruction.
- *Algorithm* is an integer to select which HEXMATE algorithm to use to calculate the checksum result. A list of selectable algorithms are given in Table 6-9. If unspecified, the default checksum algorithm used is 8 bit addition (1).

A typical example of the use of the checksum option is:

```
-CK=0-1FFF@2FFE+2100w2
```

This will calculate a checksum over the range `0-1FFFh` and program the checksum result at address `2FFEh`. The checksum value will be offset by `2100h`. The result will be two bytes wide.

TABLE 6-9: `HEXMATE` **CHECKSUM ALGORITHM SELECTION**

Selector	Algorithm description
-4	Subtraction of 32 bit values from initial value
-3	Subtraction of 24 bit values from initial value
-2	Subtraction of 16 bit values from initial value
-1	Subtraction of 8 bit values from initial value
1	Addition of 8 bit values from initial value
2	Addition of 16 bit values from initial value
3	Addition of 24 bit values from initial value
4	Addition of 32 bit values from initial value
7	Fletcher's checksum (8 bit)
8	Fletcher's checksum (16 bit)

6.6.1.6   -FILL

The `-FILL` option is used for filling unused memory locations with a known value. The usage of this option is:

`-FILL=Code@Start-End[,data]`

where:

- *Code* is the opcode that will be assigned to unused locations in memory. Multi-byte codes should be entered in little endian order.
- *Start* and *End* specify the address range over which this fill will apply.
- The `data` flag will specify that only records within the range that contain data will be filled. The default is to fill all records in the range.

For example:

`-FILL=3412@0-1FFF,data`

will program opcode `1234h` in all unused addresses from program memory address `0` to `1FFFh` (Note the endianism).

This option accepts whole bytes of hexadecimal data from 1 to 8 bytes in length.

If the data flag has been specified, `HEXMATE` will only perform ROM filling to records that actually contain data. This means that these records will be padded out to the default data record length or the width specified in the `-FORMAT` option. Records will also begin on addresses which are multiples of the data record length used. The default data record length is 16 bytes. This facility is particularly useful or is a requirement for some bootloaders that expect that all data records will be of a particular length and address alignment.

6.6.1.7   -FIND

This option is used to detect and log occurrences of an opcode or partial code sequence. The usage of this option is:

`-FIND=Findcode [mMask]@Start-End [/Align][w][t"Title"]`

where:

- *Findcode* is the hexadecimal code sequence to search for and is entered in little endian byte order.
- *Mask* is optional. It specifies a bit mask applied over the *Findcode* value to allow a less restrictive search. It is entered in little endian byte order.

- *Start* and *End* limit the address range to search.
- *Align* is optional. It specifies that a code sequence can only match if it begins on an address which is a multiple of this value.
- *w*, if present, will cause HEXMATE to issue a warning whenever the code sequence is detected.
- *Title* is optional. It allows a title to be given to this code sequence. Defining a title will make log-reports and messages more descriptive and more readable. A title will not affect the actual search results.

Here are some examples.

The option -FIND=3412@0-7FFF/2w will detect the code sequence 1234h when aligned on a 2 (two) byte address boundary, between 0h and 7FFFh. w indicates that a warning will be issued each time this sequence is found.

In this next example, -FIND=3412M0F00@0-7FFF/2wt"ADDXY", the option is the same as in last example but the code sequence being matched is masked with 000Fh, so HEXMATE will search for any of the opcodes 123xh, where x is any digit. If a byte-mask is used, is must be of equal byte-width to the opcode it is applied to. Any messaging or reports generated by HEXMATE will refer to this opcode by the name, *ADDXY* as this was the title defined for this search.

If HEXMATE is generating a log file, it will contain the results of all searches. -FIND accepts whole bytes of HEX data from 1 to 8 bytes in length. Optionally, -FIND can be used in conjunction with REPLACE or DELETE (as described below).

6.6.1.8   -FIND...,DELETE

If the DELETE form of the -FIND option is used, any matching sequences will be removed. This function should be used with extreme caution and is not normally recommended for removal of executable code.

6.6.1.9   -FIND...,REPLACE

If the REPLACE form of the -FIND option is used, any matching sequences will be replaced, or partially replaced, with new codes. The usage for this sub-option is:

-FIND...,REPLACE=*Code* [m*Mask*]

where:

- *Code* is a little endian hexadecimal code to replace the sequences that match the -FIND criteria.
- *Mask* is an optional bit mask to specify which bits within *Code* will replace the code sequence that has been matched. This may be useful if, for example, it is only necessary to modify 4 bits within a 16-bit instruction. The remaining 12 bits can masked and be left unchanged.

6.6.1.10   -FORMAT

The -FORMAT option can be used to specify a particular variant of INHX format or adjust maximum record length. The usage of this option is:

-FORMAT=*Type* [,*Length*]

where:

- *Type* specifies a particular INHX format to generate.
- *Length* is optional and sets the maximum number of bytes per data record. A valid length is between 1 and 16, with 16 being the default.

Consider the case of a bootloader trying to download an INHX32 file which fails because it cannot process the extended address records which are part of the INHX32 standard. You know that this bootloader can only program data addressed within the

range 0 to 64k, and that any data in the HEX file outside of this range can be safely disregarded. In this case, by generating the HEX file in INHX8M format the operation might succeed. The `HEXMATE` option to do this would be `-FORMAT=INHX8M`.

Now consider if the same bootloader also required every data record to contain eight bytes of data, no more, no less. This is possible by combining the `-FORMAT` with `-FILL` options. Appropriate use of `-FILL` can ensure that there are no gaps in the data for the address range being programmed. This will satisfy the minimum data length requirement. To set the maximum length of data records to eight bytes, just modify the previous option to become `-FORMAT=INHX8M,8`.

The possible types that are supported by this option are listed in Table 6-10. Note that INHX032 is not an actual INHX format. Selection of this type generates an INHX32 file but will also initialize the upper address information to zero. This is a requirement of some device programmers.

TABLE 6-10:   INHX TYPES USED IN -FORMAT OPTION

Type	Description
INHX8M	Cannot program addresses beyond 64K
INHX32	Can program addresses beyond 64K with extended linear address records
INHX032	INHX32 with initialization of upper address to zero

6.6.1.11   -HELP

Using `-HELP` will list all `HEXMATE` options. By entering another `HEXMATE` option as a parameter of `-HELP` will show a detailed help message for the given option. For example:

`-HELP=string`

will show additional help for the `-STRING HEXMATE` option.

6.6.1.12   -LOGFILE

The `-LOGFILE` option saves HEX file statistics to the named file. For example:

`-LOGFILE=output.log`

will analyze the HEX file that `HEXMATE` is generating and save a report to a file named `output.log`.

6.6.1.13   -MASK

Use this option to logically AND a memory range with a particular bitmask. This is used to ensure that the unimplemented bits in program words (if any) are left blank. The usage of this option is as follows:

`-MASK=hexcode@start-end`

Where `hexcode` is a hexadecimal value that will be ANDed with data within the `start` to `end` address range. Multibyte mask values can be entered in little endian byte order.

6.6.1.14   -OFILE

The generated Intel HEX output will be created in this file. For example:

`-Oprogram.hex`

will save the resultant output to `program.hex`. The output file can take the same name as one of its input files, but by doing so it will replace the input file entirely.

6.6.1.15   -SERIAL

This option will store a particular HEX value at a fixed address. The usage of this option is:

`-SERIAL=`*Code* `[+/-`*Increment*`]@`*Address* `[+/-`*Interval*`][r`*Repetitions*`]`

where:

* *Code* is a hexadecimal value to store and is entered in little endian byte order.
* *Increment* is optional and allows the value of *Code* to change by this value with each repetition (if requested).
* *Address* is the location to store this code, or the first repetition thereof.
* *Interval* is optional and specifies the address shift per repetition of this code.
* *Repetitions* is optional and specifies the number of times to repeat this code.

For example:

`-SERIAL=000001@EFFE`

will store HEX code `00001h` to address `EFFEh`.

Another example:

`-SERIAL=0000+2@1000+10r5`

will store 5 codes, beginning with value `0000` at address `1000h`. Subsequent codes will appear at address intervals of `+10h` and the code value will change in increments of `+2h`.

6.6.1.16   -SIZE

Using the `-SIZE` option will report the number of bytes of data within the resultant HEX image to standard output. The size will also be recorded in the log file if one has been requested.

6.6.1.17   -STRING

The `-STRING` option will embed an ASCII string at a fixed address. The usage of this option is:

`-STRING@`*Address* `[t`*Code*`]=`*"Text"*

where:

* *Address* is the location to store this string.
* *Code* is optional and allows a byte sequence to trail each byte in the string. This can allow the bytes of the string to be encoded within an instruction.
* *Text* is the string to convert to ASCII and embed.

For example:

`-STRING@1000="My favorite string"`

will store the ASCII data for the string, `My favorite string` (including the nul character terminator) at address `1000h`.

And again:

`-STRING@1000t34="My favorite string"`

will store the same string with every byte in the string being trailed with the HEX code `34h`.

6.6.1.18   -STRPACK

This option performs the same function as `-STRING` but with two important differences. Firstly, only the lower seven bits from each character are stored. Pairs of 7 bit characters are then concatenated and stored as a 14 bit word rather than in separate bytes. This is known as string packing. This is usually only useful for devices where program space is addressed as 14 bit words (PIC10/12/16 devices). The second difference is that `-STRING`'s `t` specifier is not applicable with the `-STRPACK` option.

Chapter 7. Library Functions

The functions and preprocessor macros within the standard compiler library are alphabetically listed in this chapter.

The synopsis indicates the header file in which a declaration or definition for function or macro is found. It also shows the function prototype for functions, or the equivalent prototype for macros.

__CONFIG

Synopsis

```
#include <htc.h>

__CONFIG(data)
```

Description

This macro is used to program the configuration fuses that set the device's operating modes.

The macro assumes the argument is a16-bit value, which will be used to program the configuration bits.

16-bit masks have been defined to describe each programmable attribute available on each device. These masks can be found in the chip-specific header files included via `<htc.h>`.

Multiple attributes can be selected by ANDing them together.

Example

```
#include <htc.h>

__CONFIG(RC & UNPROTECT)

void
main (void)
{
}
```

See also

`__EEPROM_DATA()`, `__IDLOC()`, `__IDLOC7()`

__DELAY_MS, __DELAY_US

Synopsis

```
__delay_ms(x)  // request a delay in milliseconds
__delay_us(x)  // request a delay in microseconds
```

Description

As it is often more convenient request a delay in time-based terms rather than in cycle counts, the macros __delay_ms(*x*) and __delay_us(*x*) are provided. These macros simply wrap around _delay(*n*) and convert the time based request into instruction cycles based on the system frequency. In order to achieve this, these macros require the prior definition of preprocessor symbol _XTAL_FREQ. This symbol should be defined as the oscillator frequency (in Hertz) used by the system.

An error will result if these macros are used without defining oscillator frequency symbol or if the delay period requested is too large.

See also

_delay()

__EEPROM_DATA

Synopsis

```
#include <htc.h>

__EEPROM_DATA(a,b,c,d,e,f,g,h)
```

Description

This macro is used to store initial values into the device's EEPROM registers at the time of programming.

The macro must be given blocks of 8 bytes to write each time it is called, and can be called repeatedly to store multiple blocks.

__EEPROM_DATA() will begin writing to EEPROM address zero, and will auto-increment the address written to by 8, each time it is used.

Example

```
#include <htc.h>

__EEPROM_DATA(0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07)
__EEPROM_DATA(0x08,0x09,0x0A,0x0B,0x0C,0x0D,0x0E,0x0F)

void
main (void)
{
}
```

See also

__CONFIG()

__IDLOC

Synopsis

```
#include <htc.h>

__IDLOC(x)
```

Description

This macro places data into the device's special locations outside of addressable memory reserved for ID. This would be useful for storage of serial numbers etc.

The macro will attempt to write 4 nibbles of data to the 4 locations reserved for ID purposes.

Example

```
#include <htc.h>

/* will store 1, 5, F and 0 in the ID registers */
__IDLOC(15F0);

void
main (void)
{
}
```

See also

```
__IDLOC7(),__CONFIG()
```

__IDLOC7

Synopsis

```
#include <htc.h>

__IDLOC7(a,b,c,d)
```

Description

This macro places data into the device's special locations outside of addressable memory reserved for ID. This would be useful for storage of serial numbers etc.

The macro will attempt to write 7 bits of data to each of the 4 locations reserved for ID purposes.

Example

```
#include <htc.h>

/* will store 7Fh, 70, 1 and 5Ah in the ID registers */
__IDLOC(0x7F,70,1,0x5A);

void
main (void)
{
}
```

Note

Not all devices permit 7 bit programming of the ID locations. Refer to the device data sheet to see whether this macro can be used on your particular device.

See also

```
__IDLOC(), __CONFIG()
```

_DELAY()

Synopsis

```
#include <htc.h>

void _delay(unsigned long cycles);
```

Description

This is an inline function that is expanded by the code generator. When called, this routine expands to an inline assembly delay sequence. The sequence will consist of code that delays for the number of cycles that is specified as argument. The argument must be a literal constant.

An error will result if the delay period requested is too large. For very large delays, call this function multiple times.

Example

```
#include <htc.h>

void
main (void)
{
  control |= 0x80;
  _delay(10);    // delay for 10 cycles
  control &= 0x7F;
}
```

See Also

```
__delay_us(), __delay_ms()
```

ABS

Synopsis

```
#include <stdlib.h>

int abs (int j)
```

Description

The `abs()` function returns the absolute value of `j`.

Example

```
#include <stdio.h>
#include <stdlib.h>

void
main (void)
{
    int a = -5;

    printf(" absolute value of %d is %d\n" a, abs(a));
}
```

See Also

`labs()`, `fabs()`

Return Value

The absolute value of `j`.

ACOS

Synopsis

```
#include <math.h>

double acos (double f)
```

Description

The `acos()` function implements the inverse of cos(), i.e. it is passed a value in the range -1 to +1, and returns an angle in radians whose cosine is equal to that value.

Example

```
#include <math.h>
#include <stdio.h>

/* Print acos() values for -1 to 1 in degrees. */

void
main (void)
{
    float i, a;

    for(i = -1.0; i < 1.0 ; i += 0.1) {
        a = acos(i)*180.0/3.141592;
        printf("(%f) = %f degrees\n" i, a);
    }
}
```

See Also

`sin(), cos(), tan(), asin(), atan(), atan2()`

Return Value

An angle in radians, in the range 0 to π

ASCTIME

Synopsis

```
#include <time.h>

char * asctime (struct tm * t)
```

Description

The `asctime()` function takes the time broken down into the `struct tm` structure, pointed to by its argument, and returns a 26 character string describing the current date and time in the format:

Sun Sep 16 01:03:52 1973\n\0

Note the newline at the end of the string. The width of each field in the string is fixed. The example gets the current time, converts it to a `struct tm` pointer with `localtime()`, it then converts this to ASCII and prints it. The `time()` function will need to be provided by the user (see `time()` for details).

Example

```
#include <stdio.h>
#include <time.h>

void
main (void)
{
    time_t clock;
    struct tm * tp;

    time(&clock);
    tp = localtime(&clock);
    printf("s" asctime(tp));
}
```

See Also

`ctime(), gmtime(), localtime(), time()`

Return Value

A pointer to the string.

Note

The example will require the user to provide the `time()` routine as it cannot be supplied with the compiler. See time() for more details.

ASIN

Synopsis

```
#include <math.h>

double asin (double f)
```

Description

The `asin()` function implements the converse of `sin()`, i.e. it is passed a value in the range -1 to +1, and returns an angle in radians whose sine is equal to that value.

Example

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    float i, a;

    for(i = -1.0; i < 1.0 ; i += 0.1) {
        a = asin(i)*180.0/3.141592;
        printf("(%f) = %f degrees\n" i, a);
    }
}
```

See Also

`sin()`, `cos()`, `tan()`, `acos()`, `atan()`, `atan2()`

Return Value

An angle in radians, in the range - π

ASSERT

Synopsis

```
#include <assert.h>

void assert (int e)
```

Description

This macro is used for debugging purposes; the basic method of usage is to place assertions liberally throughout your code at points where correct operation of the code depends upon certain conditions being true initially. An `assert()` routine may be used to ensure at run time that an assumption holds true. For example, the following statement asserts that the pointer tp is not equal to NULL:

```
assert(tp);
```

If at run time the expression evaluates to false, the program will abort with a message identifying the source file and line number of the assertion, and the expression used as an argument to it. A fuller discussion of the uses of `assert()` is impossible in limited space, but it is closely linked to methods of proving program correctness.

Example

```
void
ptrfunc (struct xyz * tp)
{
    assert(tp != 0);
}
```

Note

When required for ROM based systems, the underlying routine `_fassert(...)` will need to be implemented by the user.

ATAN

Synopsis

```
#include <math.h>

double atan (double x)
```

Description

This function returns the arc tangent of its argument, i.e. it returns an angle e in the range - π

Example

```
#include <stdio.h>
#include <math.h>

void
main (void)
{
    printf("f\n" atan(1.5));
}
```

See Also

`sin(), cos(), tan(), asin(), acos(), atan2()`

Return Value

The arc tangent of its argument.

ATAN2

Synopsis

```
#include <math.h>

double atan2 (double x, double x)
```

Description

This function returns the arc tangent of `y/x`.

Example

```
#include <stdio.h>
#include <math.h>

void
main (void)
{
    printf("f\n" atan2(10.0, -10.0));
}
```

See Also

`sin(), cos(), tan(), asin(), acos(), atan()`

Return Value

The arc tangent of `y/x`.

ATOF

Synopsis

```
#include <stdlib.h>

double atof (const char * s)
```

Description

The `atof()` function scans the character string passed to it, skipping leading blanks. It then converts an ASCII representation of a number to a double. The number may be in decimal, normal floating point or scientific notation.

Example

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    char buf[80];
    double i;

    gets(buf);
    i = atof(buf);
    printf(" %s: converted to %f\n" buf, i);
}
```

See Also

`atoi()`, `atol()`, `strtod()`

Return Value

A double precision floating point number. If no number is found in the string, 0.0 will be returned.

ATOI

Synopsis

```
#include <stdlib.h>

int atoi (const char * s)
```

Description

The `atoi()` function scans the character string passed to it, skipping leading blanks and reading an optional sign. It then converts an ASCII representation of a decimal number to an integer.

Example

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    char buf[80];
    int i;

    gets(buf);
    i = atoi(buf);
    printf(" %s: converted to %d\n" buf, i);
}
```

See Also

`xtoi()`, `atof()`, `atol()`

Return Value

A signed integer. If no number is found in the string, 0 will be returned.

ATOL

Synopsis

```
#include <stdlib.h>

long atol (const char * s)
```

Description

The `atol()` function scans the character string passed to it, skipping leading blanks. It then converts an ASCII representation of a decimal number to a long integer.

Example

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    char buf[80];
    long i;

    gets(buf);
    i = atol(buf);
    printf(" %s: converted to %ld\n" buf, i);
}
```

See Also

`atoi()`, `atof()`

Return Value

A long integer. If no number is found in the string, 0 will be returned.

BSEARCH

Synopsis

```
#include <stdlib.h>

void * bsearch (const void * key, void * base, size_t n_memb,
size_t size, int (*compar)(const void *, const void *))
```

Description

The `bsearch()` function searches a sorted array for an element matching a particular key. It uses a binary search algorithm, calling the function pointed to by `compar` to compare elements in the array.

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

struct value {
    char name[40];
    int value;
} values[100];

int
val_cmp (const void * p1, const void * p2)
{
    return strcmp(((const struct value *)p1)->name,
                  ((const struct value *)p2)->name);
}

void
main (void)
{
    char inbuf[80];
    int i;
```

```
            struct value * vp;

            i = 0;
            while(gets(inbuf)) {
                sscanf(inbuf,"s %d" values[i].name, &values[i].value);
                i++;
            }
            qsort(values, i, sizeof values[0], val_cmp);
            vp = bsearch("" values, i, sizeof values[0], val_cmp);
            if(!vp)
                printf(" 'fred' was not found\n";
            else
                printf(" 'fred' has value %d\n" vp->value);
    }
```

See Also

```
qsort()
```

Return Value

A pointer to the matched array element (if there is more than one matching element, any of these may be returned). If no match is found, a null pointer is returned.

Note

The comparison function must have the correct prototype.

CEIL

Synopsis

```
#include <math.h>
```

```
double ceil (double f)
```

Description

This routine returns the smallest whole number not less than `f`.

Example

```
#include <stdio.h>
#include <math.h>

void
main (void)
{
    double j;

    scanf("lf" &j);
    printf(" ceiling of %lf is %lf\n" j, ceil(j));
}
```

CGETS

Synopsis

```
#include <conio.h>

char * cgets (char * s)
```

Description

The cgets() function will read one line of input from the console into the buffer passed as an argument. It does so by repeated calls to getche(). As characters are read, they are buffered, with backspace deleting the previously typed character, and ctrl-U deleting the entire line typed so far. Other characters are placed in the buffer, with a carriage return or line feed (newline) terminating the function. The collected string is null terminated.

Example

```
#include <conio.h>
#include <string.h>

char buffer[80];

void
main (void)
{
    for(;;) {
        cgets(buffer);
        if(strcmp(buffer, "" == 0)
            break;
        cputs(" 'exit' to finish\n";
    }
}
```

See Also

getch(), getche(), putch(), cputs()

Return Value

The return value is the character pointer passed as the sole argument.

CLRWDT

Synopsis

```
#include <htc.h>

CLRWDT();
```

Description

This macro is used to clear the device's internal watchdog timer.

Example

```
#include <htc.h>

void
main (void)
{
    WDTCON=1;
      /* enable the WDT */

    CLRWDT();
}
```

COS

Synopsis

```
#include <math.h>

double cos (double f)
```

Description

This function yields the cosine of its argument, which is an angle in radians. The cosine is calculated by expansion of a polynomial series approximation.

Example

```
#include <math.h>
#include <stdio.h>

#define C 3.141592/180.0

void
main (void)
{
    double i;

    for(i = 0 ; i <= 180.0 ; i += 10)
        printf("(%3.0f) = %f, cos = %f\n" i, sin(i*C), cos(i*C));
}
```

See Also

```
sin(), tan(), asin(), acos(), atan(), atan2()
```

Return Value

A double in the range -1 to +1.

COSH, SINH, TANH

Synopsis

```
#include <math.h>

double cosh (double f)
double sinh (double f)
double tanh (double f)
```

Description

These functions are the implement hyperbolic equivalents of the trigonometric functions; `cos()`, `sin()` and `tan()`.

Example

```
#include <stdio.h>
#include <math.h>

void
main (void)
{
    printf("f\n" cosh(1.5));
    printf("f\n" sinh(1.5));
    printf("f\n" tanh(1.5));
}
```

Return Value

The function `cosh()` returns the hyperbolic cosine value.

The function `sinh()` returns the hyperbolic sine value.

The function `tanh()` returns the hyperbolic tangent value.

CPUTS

Synopsis

```
#include <conio.h>

void cputs (const char * s)
```

Description

The `cputs()` function writes its argument string to the console, outputting carriage returns before each newline in the string. It calls `putch()` repeatedly. On a hosted system `cputs()` differs from `puts()` in that it writes to the console directly, rather than using file I/O. In an embedded system `cputs()` and `puts()` are equivalent.

Example

```
#include <conio.h>
#include <string.h>

char buffer[80];

void
main (void)
{
    for(;;) {
        cgets(buffer);
        if(strcmp(buffer, "" == 0)
            break;
        cputs(" 'exit' to finish\n";
    }
}
```

See Also

`cputs()`, `puts()`, `putch()`

CTIME

Synopsis

```
#include <time.h>

char * ctime (time_t * t)
```

Description

The `ctime()` function converts the time in seconds pointed to by its argument to a string of the same form as described for `asctime()`. Thus the example program prints the current time and date.

Example

```
#include <stdio.h>
#include <time.h>

void
main (void)
{
    time_t clock;

    time(&clock);
    printf("s" ctime(&clock));
}
```

See Also

`gmtime()`, `localtime()`, `asctime()`, `time()`

Return Value

A pointer to the string.

Note

The example will require the user to provide the `time()` routine as one cannot be supplied with the compiler. See `time()` for more detail.

DEVICE_ID_READ()

Synopsis

```
#include <htc.h>

unsigned int device_id_read(void);
```

Description

This function returns the device ID code that is factory-programmed into the chip. This code can be used to identify the device and its revision number.

Example

```
#include <htc.h>

void
main (void)
{
    unsigned int   id_value;
    unsigned int   device_code;
    unsigned char revision_no;

    id_value = device_id_read();
        /* lower 5 bits represent revision number
         * upper 11 bits identify device */
    device_code = (id_value >> 5);
    revision_no = (unsigned char)(id_value & 0x1F);

}
```

See Also

`flash_read()`, `config_read()`

Return Value

`device_id_read()` returns the 16-Bit factory-programmed device id code used to identify the device type and its revision number.

Note

The `device_id_read()` is applicable only to those devices which are capable of reading their own program memory.

DI, EI

Synopsis

```
#include <htc.h>

void ei (void)
void di (void)
```

Description

The `di()` and `ei()` routines disable and re-enable interrupts respectively. These are implemented as macros. The example shows the use of `ei()` and `di()` around access to a long variable that is modified during an interrupt. If this was not done, it would be possible to return an incorrect value, if the interrupt occurred between accesses to successive words of the count value.

The `ei()` macro should never be called in an interrupt function, and there is no need to call `di()` in an interrupt function.

Example

```
#include <htc.h>

long count;

void
interrupt tick (void)
{
    count++;
}

long
getticks (void)
{
    long val;    /* Disable interrupts around access
                    to count, to ensure consistency.*/
    di();
    val = count;
    ei();
    return val;
}
```

DIV

Synopsis

```
#include <stdlib.h>

div_t div (int numer, int demon)
```

Description

The `div()` function computes the quotient and remainder of the numerator divided by the denominator.

Example

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    div_t x;

    x = div(12345, 66);
    printf(" = %d, remainder = %d\n" x.quot, x.rem);
}
```

See Also

```
udiv(), ldiv(), uldiv()
```

Return Value

Returns the quotient and remainder into the `div_t` structure.

EEPROM_READ, EEPROM_WRITE

Synopsis

```
#include <htc.h>

unsigned char eeprom_read (unsigned int address);
void eeprom_write (unsigned int address, unsigned char value);
```

Description

These functions allow access to the on-chip eeprom (when present). The eeprom is not in the directly-accessible memory space and a special byte sequence is loaded to the eeprom control registers to access this memory. Writing a value to the eeprom is a slow process and the `eeprom_write()` function polls the appropriate registers to ensure that any previous writes have completed before writing the next datum.

Reading data is completed in the one cycle and no polling is necessary to check for a read completion.

Example

```
#include <htc.h>

void
main (void)
{
    unsigned char data;
    unsigned int address = 0x0010;

    data=eeprom_read(address);
    eeprom_write(address, data);
}
```

See Also

```
flash_erase(), flash_read(), flash_write()
```

Note

The high and low priority interrupt are disabled during sensitive sequences required to access EEPROM. Interrupts are restored after the sequence has completed. `eeprom_write()` will clear the EEIF hardware flag before returning.

Both `eeprom_read()` and `eeprom_write()` are available in a similar macro form. The essential difference between the macro and function implementations is that `EEPROM_READ()`, the macro, does not test nor wait for any prior write operations to complete.

EVAL_POLY

Synopsis

```
#include <math.h>

double eval_poly (double x, const double * d, int n)
```

Description

The `eval_poly()` function evaluates a polynomial, whose coefficients are contained in the array `d`, at `x`, for example:

`y = x*x*d2 + x*d1 + d0.`

The order of the polynomial is passed in `n`.

Example

```
#include <stdio.h>
#include <math.h>

void
main (void)
{
    double x, y;
    double d[3] = {1.1, 3.5, 2.7};

    x = 2.2;
    y = eval_poly(x, d, 2);
    printf(" polynomial evaluated at %f is %f\n" x, y);
}
```

Return Value

A double value, being the polynomial evaluated at `x`.

EXP

Synopsis

```
#include <math.h>

double exp (double f)
```

Description

The `exp()` routine returns the exponential function of its argument, i.e. `e` to the power of `f`.

Example

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    double f;

    for(f = 0.0 ; f <= 5 ; f += 1.0)
        printf(" to %1.0f = %f\n" f, exp(f));
}
```

See Also

`log(), log10(),pow()`

FABS

Synopsis

```
#include <math.h>

double fabs (double f)
```

Description

This routine returns the absolute value of its double argument.

Example

```
#include <stdio.h>
#include <math.h>

void
main (void)
{
    printf("f %f\n" fabs(1.5), fabs(-1.5));
}
```

See Also

`abs(),labs()`

FLASH_COPY

Synopsis

```
#include <htc.h>

void flash_copy(const unsigned char * source_addr,
    unsigned char length, unsigned short dest_addr);
```

Description

This utility function is useful for copying a large section of memory to a new location in Flash memory.

Note it is only applicable to those devices which have an internal set of Flash buffer registers.

When the function is called, it needs to be supplied with a `const pointer` to the source address of the data to copy. The pointer may point to a valid address in either RAM or Flash memory.

A length parameter must be specified to indicate the number of words of the data to be copied.

Finally the Flash address where this data is destined must be specified.

Example

```
#include <htc.h>

const unsigned char ROMSTRING[] = ""

void
main (void){
    const unsigned char * ptr = &ROMSTRING[0];
    flash_copy( ptr, 5, 0x70 );
}
```

See Also

`EEPROM_READ()`, `EEPROM_WRITE()`, `FLASH_READ()`, `FLASH_WRITE()`

Note

This function is only applicable to those devices which use internal buffer registers when writing to Flash.

Ensure that the function does not attempt to overwrite the section of program memory from which it is currently executing, and extreme caution must be exercised if modifying code at the device's reset or interrupt vectors. A reset or interrupt must not be triggered while this sector is in erasure.

FLASH_ERASE(), FLASH_READ()

Synopsis

```
#include <htc.h>

void flash_erase (unsigned short addr);
unsigned int flash_read (unsigned short addr);
```

Description

These functions allow access to the Flash memory of the microcontroller (if supported).

Reading from the Flash memory can be done one word at a time with use of the `flash_read()` function. `flash_read()` returns the data value found at the specified word address in Flash memory.

Entire sectors of 32 words can be restored to an unprogrammed state (value=`FF`) with use of the `flash_erase()` function. Specifying an address to the `flash_erase()` function, will erase all 32 words in the sector that contains the given address.

Example

```
#include <htc.h>

void
main (void)
{
    unsigned int data;
    unsigned short address=0x1000;

    data = flash_read(address);

    flash_erase(address);

}
```

Return Value

`flash_read()` returns the data found at the given address, as an `unsigned int`.

Note

The functions `flash_erase()` and `flash_read()` are only available on those devices that support such functionality.

FMOD

Synopsis

```
#include <math.h>

double fmod (double x, double y)
```

Description

The function `fmod` returns the remainder of x/y as a floating point quantity.

Example

```
#include <math.h>

void
main (void)
{
    double rem, x;

    x = 12.34;
    rem = fmod(x, 2.1);
}
```

Return Value

The floating-point remainder of x/y.

FLOOR

Synopsis

```
#include <math.h>

double floor (double f)
```

Description

This routine returns the largest whole number not greater than `f`.

Example

```
#include <stdio.h>
#include <math.h>

void
main (void)
{
    printf("f\n" floor( 1.5 ));
    printf("f\n" floor( -1.5));
}
```

FREXP

Synopsis

```
#include <math.h>

double frexp (double f, int * p)
```

Description

The `frexp()` function breaks a floating point number into a normalized fraction and an integral power of 2. The integer is stored into the `int` object pointed to by `p`. Its return value x is in the interval (0.5, 1.0) or zero, and `f` equals x times 2 raised to the power stored in `*p`. If `f` is zero, both parts of the result are zero.

Example

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    double f;
    int i;

    f = frexp(23456.34, &i);
    printf(".34 = %f * 2^%d\n" f, i);
}
```

See Also

```
ldexp()
```

FTOA

Synopsis

```
#include <stdlib.h>

char * ftoa (float f, int * status)
```

Description

The function `ftoa` converts the contents of `f` into a string which is stored into a buffer which is then return.

Example

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    char * buf;
    float input = 12.34;
    int status;
    buf = ftoa(input, &status);
    printf(" buffer holds %s\n" buf);
}
```

See Also

`strtol()`, `itoa()`, `utoa()`, `ultoa()`

Return Value

This routine returns a reference to the buffer into which the result is written.

GETCHAR

Synopsis

```
#include <stdio.h>

int getchar (void)
```

Description

The `getchar()` routine is a `getc(stdin)` operation. It is a macro defined in `stdio.h`. Note that under normal circumstances `getchar()` will NOT return unless a carriage return has been typed on the console. To get a single character immediately from the console, use the function `getch()`.

Example

```
#include <stdio.h>

void
main (void)
{
    int c;

    while((c = getchar()) != EOF)
        putchar(c);
}
```

See Also

`getc(), fgetc(), freopen(), fclose()`

Note

This routine is not usable in a ROM based system.

GETS

Synopsis

```
#include <stdio.h>

char * gets (char * s)
```

Description

The `gets()` function reads a line from standard input into the buffer at `s`, deleting the newline (c.f. `fgets()`). The buffer is null terminated. In an embedded system, `gets()` is equivalent to `cgets()`, and results in `getche()` being called repeatedly to get characters. Editing (with backspace) is available.

Example

```
#include <stdio.h>

void
main (void)
{
    char buf[80];

    printf(" a line: ";
    if(gets(buf))
        puts(buf);
}
```

See Also

`fgets(), freopen(), puts()`

Return Value

It returns its argument, or NULL on end-of-file.

GET_CAL_DATA

Synopsis

```
#include <htc.h>

double get_cal_data (const unsigned char * code_ptr)
```

Description

This function returns the 32-bit floating point calibration data from the PIC MCU 14000 calibration space. Only use this function to access `KREF`, `KBG`, `VHTHERM` and `KTC` (that is, the 32-bit floating point parameters). `FOSC` and `TWDT` can be accessed directly as they are bytes.

Example

```
#include <htc.h>

void
main (void)
{
    double x;
    unsigned char y;

        /* Get the slope reference ratio. */
    x = get_cal_data(KREF);

        /* Get the WDT time-out. */
    y =TWDT;
}
```

Return Value

The value of the calibration parameter

Note

This function can only be used on the PIC 14000.

GMTIME

Synopsis

```
#include <time.h>

struct tm * gmtime (time_t * t)
```

Description

This function converts the time pointed to by `t` which is in seconds since 00:00:00 on Jan 1, 1970, into a broken down time stored in a structure as defined in `time.h`. The structure is defined in the 'Data Types' section.

Example

```
#include <stdio.h>
#include <time.h>

void
main (void)
{
    time_t clock;
    struct tm * tp;

    time(&clock);
    tp = gmtime(&clock);
    printf("'s %d in London\n" tp->tm_year+1900);
}
```

See Also

`ctime(), asctime(), time(), localtime()`

Return Value

Returns a structure of type `tm`.

Note

The example will require the user to provide the time() routine as one cannot be supplied with the compiler. See time() for more detail.

ISALNUM, ISALPHA, ISDIGIT, ISLOWER ET. AL.

Synopsis

```
#include <ctype.h>

int isalnum (char c)
int isalpha (char c)
int isascii (char c)
int iscntrl (char c)
int isdigit (char c)
int islower (char c)
int isprint (char c)
int isgraph (char c)
int ispunct (char c)
int isspace (char c)
int isupper (char c)
int isxdigit(char c)
```

Description

These macros, defined in ctype.h, test the supplied character for membership in one of several overlapping groups of characters. Note that all except isascii() are defined for c, if isascii(c) is true or if c = EOF.

isalnum(c)	c is in 0-9 or a-z or A-Z
isalpha(c)	c is in A-Z or a-z
isascii(c)	c is a 7 bit ascii character
iscntrl(c)	c is a control character
isdigit(c)	c is a decimal digit
islower(c)	c is in a-z
isprint(c)	c is a printing char
isgraph(c)	c is a non-space printable character
ispunct(c)	c is not alphanumeric
isspace(c)	c is a space, tab or newline
isupper(c)	c is in A-Z
isxdigit(c)	c is in 0-9 or a-f or A-F

Example

```
#include <ctype.h>
#include <stdio.h>

void
main (void)
{
    char buf[80];
    int i;

    gets(buf);
    i = 0;
    while(isalnum(buf[i]))
        i++;
    buf[i] = 0;
    printf("%s' is the word\n" buf);
}
```

See Also

```
toupper(), tolower(), toascii()
```

ISDIG

Synopsis

```
#include <ctype.h>

int isdig (int c)
```

Description

The isdig() function tests the input character c to see if is a decimal digit (0 – 9) and returns true is this is the case; false otherwise.

Example

```
#include <ctype.h>

void
main (void)
{
    char buf[] = ""
    if(isdig(buf[0]))
        printf(" type detected\n";
}
```

See Also

isdigit() (listed under isalnum())

Return Value

Zero if the character is a decimal digit; a non-zero value otherwise.

ITOA

Synopsis

```
#include <stdlib.h>

char * itoa (char * buf, int val, int base)
```

Description

The function itoa converts the contents of val into a string which is stored into buf. The conversion is performed according to the radix specified in base. buf is assumed to reference a buffer which has sufficient space allocated to it.

Example

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    char buf[10];
    itoa(buf, 1234, 16);
    printf(" buffer holds %s\n" buf);
}
```

See Also

```
strtol(), utoa(), ltoa(), ultoa()
```

Return Value

This routine returns a copy of the buffer into which the result is written.

LABS

Synopsis

```
#include <stdlib.h>

int labs (long int j)
```

Description

The `labs()` function returns the absolute value of long value `j`.

Example

```
#include <stdio.h>
#include <stdlib.h>

void
main (void)
{
    long int a = -5;

    printf(" absolute value of %ld is %ld\n" a, labs(a));
}
```

See Also

```
abs()
```

Return Value

The absolute value of `j`.

LDEXP

Synopsis

```
#include <math.h>

double ldexp (double f, int i)
```

Description

The `ldexp()` function performs the inverse of `frexp()` operation; the integer `i` is added to the exponent of the floating point `f` and the resultant returned.

Example

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    double f;

    f = ldexp(1.0, 10);
    printf(".0 * 2\textasciicircum 10 = %f\n" f);
}
```

See Also

```
frexp()
```

Return Value

The return value is the integer `i` added to the exponent of the floating point value `f`.

LDIV

Synopsis

```
#include <stdlib.h>

ldiv_t ldiv (long number, long denom)
```

Description

The `ldiv()` routine divides the numerator by the denominator, computing the quotient and the remainder. The sign of the quotient is the same as that of the mathematical quotient. Its absolute value is the largest integer which is less than the absolute value of the mathematical quotient.

The `ldiv()` function is similar to the `div()` function, the difference being that the arguments and the members of the returned structure are all of type `long int`.

Example

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    ldiv_t lt;

    lt = ldiv(1234567, 12345);
    printf(" = %ld, remainder = %ld\n" lt.quot, lt.rem);
}
```

See Also

```
div(), uldiv(), udiv()
```

Return Value

Returns a structure of type `ldiv_t`

LOCALTIME

Synopsis

```
#include <time.h>

struct tm * localtime (time_t * t)
```

Description

The `localtime()` function converts the time pointed to by `t` which is in seconds since 00:00:00 on Jan 1, 1970, into a broken down time stored in a structure as defined in `time.h`. The routine `localtime()` takes into account the contents of the global integer time_zone. This should contain the number of minutes that the local time zone is westward of Greenwich. On systems where it is not possible to predetermine this value, `localtime()` will return the same result as `gmtime()`.

Example

```
#include <stdio.h>
#include <time.h>

    char * wday[] = {
        "" "" "" ""
        "" "" ""
};

void
main (void)
{
    time_t clock;
    struct tm * tp;

    time(&clock);
    tp = localtime(&clock);
    printf(" is %s\n" wday[tp->tm_wday]);
}
```

See Also

`ctime()`, `asctime()`, `time()`

Return Value

Returns a structure of type `tm`.

Note

The example will require the user to provide the `time()` routine as one cannot be supplied with the compiler. See `time()` for more detail.

LOG, LOG10

Synopsis

```
#include <math.h>

double log (double f)
double log10 (double f)
```

Description

The `log()` function returns the natural logarithm of `f`. The function `log10()` returns the logarithm to base 10 of `f`.

Example

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    double f;

    for(f = 1.0 ; f <= 10.0 ; f += 1.0)
        printf("(%1.0f) = %f\n" f, log(f));
}
```

See Also

`exp()`, `pow()`

Return Value

Zero if the argument is negative.

LONGJMP

Synopsis

```
#include <setjmp.h>

void longjmp (jmp_buf buf, int val)
```

Description

The `longjmp()` function, in conjunction with `setjmp()`, provides a mechanism for non-local goto's. To use this facility, `setjmp()` should be called with a `jmp_buf` argument in some outer level function. The call from `setjmp()` will return 0.

To return to this level of execution, `longjmp()` may be called with the same `jmp_buf` argument from an inner level of execution. Note however that the function which called `setjmp()` must still be active when `longjmp()` is called. Breach of this rule will cause disaster, due to the use of a stack containing invalid data. The `val` argument to `longjmp()` will be the value apparently returned from the `setjmp()`. This should normally be non-zero, to distinguish it from the genuine `setjmp()` call.

Example

```
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>

jmp_buf jb;

void
inner (void)
{
    longjmp(jb, 5);
}

void
main (void)
{
    int i;

    if(i = setjmp(jb)) {
        printf(" returned %d\n" i);
        exit(0);
    }
    printf(" returned 0 - good\n";
    printf(" inner...\n";
    inner();
    printf(" returned - bad!\n";
}
```

See Also

```
setjmp()
```

Return Value

The `longjmp()` routine never returns.

Note

The function which called `setjmp()` must still be active when `longjmp()` is called. Breach of this rule will cause disaster, due to the use of a stack containing invalid data.

LTOA

Synopsis

```
#include <stdlib.h>
```

```
char * ltoa (char * buf, long val, int base)
```

Description

The function `itoa` converts the contents of `val` into a string which is stored into `buf`. The conversion is performed according to the radix specified in `base`. `buf` is assumed to reference a buffer which has sufficient space allocated to it.

Example

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    char buf[10];
    utoi(buf, 12345678L, 16);
    printf(" buffer holds %s\n" buf);
}
```

See Also

`strtol(), itoa(), utoa(), ultoa()`

Return Value

This routine returns a copy of the buffer into which the result is written.

MEMCHR

Synopsis

```
#include <string.h>

void * memchr (const void * block, int val, size_t length)
```

Description

The `memchr()` function is similar to `strchr()` except that instead of searching null-terminated strings, it searches a block of memory specified by length for a particular byte. Its arguments are a pointer to the memory to be searched, the value of the byte to be searched for, and the length of the block. A pointer to the first occurrence of that byte in the block is returned.

Example

```
#include <string.h>
#include <stdio.h>

unsigned int ary[] = {1, 5, 0x6789, 0x23};

void
main (void)
{
    char * cp;

    cp = memchr(ary, 0x89, sizeof ary);
    if(!cp)
        printf(" found\n";
    else
        printf(" at offset %u\n" cp - (char *)ary);
}
```

See Also

`strchr()`

Return Value

A pointer to the first byte matching the argument if one exists; NULL otherwise.

MEMCMP

Synopsis

```
#include <string.h>

int memcmp (const void * s1, const void * s2, size_t n)
```

Description

The `memcmp()` function compares two blocks of memory, of length `n`, and returns a signed value similar to `strncmp()`. Unlike `strncmp()` the comparison does not stop on a null character.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    int buf[10], cow[10], i;

    buf[0] = 1;
    buf[2] = 4;
    cow[0] = 1;
    cow[2] = 5;
    buf[1] = 3;
    cow[1] = 3;
    i = memcmp(buf, cow, 3*sizeof(int));
    if(i < 0)
        printf(" than\n");
    else if(i > 0)
        printf(" than\n");
    else
        printf("\n");
}
```

See Also

`strncpy()`, `strncmp()`, `strchr()`, `memset()`, `memchr()`

Return Value

Returns negative one, zero or one, depending on whether `s1` points to string which is less than, equal to or greater than the string pointed to by `s2` in the collating sequence.

MEMCPY

Synopsis

```
#include <string.h>

void * memcpy (void * d, const void * s, size_t n)
```

Description

The `memcpy()` function copies `n` bytes of memory starting from the location pointed to by `s` to the block of memory pointed to by `d`. The result of copying overlapping blocks is undefined. The `memcpy()` function differs from `strcpy()` in that it copies a specified number of bytes, rather than all bytes up to a null terminator.

Example

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    char buf[80];

    memset(buf, 0, sizeof buf);
    memcpy(buf, " partial string" 10);
    printf(" = '%s'\n" buf);
}
```

See Also

`strncpy()`, `strncmp()`, `strchr()`, `memset()`

Return Value

The `memcpy()` routine returns its first argument.

MEMMOVE

Synopsis

```
#include <string.h>

void * memmove (void * s1, const void * s2, size_t n)
```

Description

The `memmove()` function is similar to the function `memcpy()` except copying of overlapping blocks is handled correctly. That is, it will copy forwards or backwards as appropriate to correctly copy one block to another that overlaps it.

See Also

`strncpy()`, `strncmp()`, `strchr()`, `memcpy()`

Return Value

The function `memmove()` returns its first argument.

MEMSET

Synopsis

```
#include <string.h>

void * memset (void * s, int c, size_t n)
```

Description

The `memset()` function fills `n` bytes of memory starting at the location pointed to by `s` with the byte `c`.

Example

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    char abuf[20];

    strcpy(abuf, " is a string";
    memset(abuf, 'x', 5);
    printf(" = '%s'\n" abuf);
}
```

See Also

`strncpy(), strncmp(), strchr(), memcpy(), memchr()`

MKTIME

Synopsis

```
#include <time.h>

time_t mktime (struct tm * tmptr)
```

Description

The `mktime()` function converts the local calendar time referenced by the tm structure pointer `tmptr` into a time being the number of seconds passed since Jan 1st 1970, or -1 if the time cannot be represented.

Example

```
#include <time.h>
#include <stdio.h>

void
main (void)
{
    struct tm birthday;

    birthday.tm_year = 1955;
    birthday.tm_mon = 2;
    birthday.tm_mday = 24;
    birthday.tm_hour = birthday.tm_min = birthday.tm_sec = 0;
    printf(" have been alive approximately %ld seconds\n"
 mktime(&birthday));
}
```

See Also

`ctime(), asctime()`

Return Value

The time contained in the `tm` structure represented as the number of seconds since the 1970 Epoch, or -1 if this time cannot be represented.

MODF

Synopsis

```
#include <math.h>

double modf (double value, double * iptr)
```

Description

The `modf()` function splits the argument `value` into integral and fractional parts, each having the same sign as `value`. For example, -3.17 would be split into the integral part (-3) and the fractional part (-0.17).

The integral part is stored as a double in the object pointed to by `iptr`.

Example

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    double i_val, f_val;

    f_val = modf( -3.17, &i_val);
}
```

Return Value

The signed fractional part of `value`.

NOP

Synopsis

```
#include <htc.h>

NOP();
```

Description

Execute `NOP` instruction here. This is often useful to finetune delays or create a handle for breakpoints. The `NOP` instruction is sometimes required during some sensitive sequences in hardware.

Example

```
#include <htc.h>

void
crude_delay(unsigned char x) {
    while(x--){
        NOP();  /* Do nothing for 3 cycles */
        NOP();
        NOP();
    }
}
```

POW

Synopsis

```
#include <math.h>

double pow (double f, double p)
```

Description

The `pow()` function raises its first argument, `f`, to the power `p`.

Example

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    double f;

    for(f = 1.0 ; f <= 10.0 ; f += 1.0)
        printf("(2, %1.0f) = %f\n" f, pow(2, f));
}
```

See Also

```
log(), log10(), exp()
```

Return Value

f to the power of `p`.

PRINTF, VPRINTF

Synopsis

```
#include <stdio.h>

int printf (const char * fmt, ...)

#include <stdio.h>
#include <stdarg.h>

int vprintf (const char * fmt, va_list va_arg)
```

Description

The printf() function is a formatted output routine, operating on stdout. The printf() routine is passed a format string, followed by a list of zero or more arguments. In the format string are conversion specifications, each of which is used to print out one of the argument list values.

Each conversion specification is of the form %m.nc where the percent symbol % introduces a conversion, followed by an optional width specification m. The n specification is an optional precision specification (introduced by the *dot*) and c is a letter specifying the type of the conversion.

A minus sign ('-') preceding m indicates left rather than right adjustment of the converted value in the field. Where the field width is larger than required for the conversion, blank padding is performed at the left or right as specified. Where right adjustment of a numeric conversion is specified, and the first digit of m is 0, then padding will be performed with zeroes rather than blanks. For integer formats, the precision indicates a minimum number of digits to be output, with leading zeros inserted to make up this number if required.

A hash character (#) preceding the width indicates that an alternate format is to be used. The nature of the alternate format is discussed below. Not all formats have alternates. In those cases, the presence of the hash character has no effect.

If the character * is used in place of a decimal constant, e.g. in the format %*d, then one integer argument will be taken from the list to provide that value. The types of conversion are:

f Floating point - m is the total width and n is the number of digits after the decimal point. If n is omitted it defaults to 6. If the precision is zero, the decimal point will be omitted unless the alternate format is specified.

e Print the corresponding argument in scientific notation. Otherwise similar to f.

g Use e or f format, whichever gives maximum precision in minimum width. Any trailing zeros after the decimal point will be removed, and if no digits remain after the decimal point, it will also be removed.

o x X u d Integer conversion - in radices 8, 16, 16, 10 and 10 respectively. The conversion is signed in the case of d, unsigned otherwise. The precision value is the total number of digits to print, and may be used to force leading zeroes. E.g. %8.4x will print at least 4 HEX digits in an 8 wide field. Preceding the key letter with an l indicates that the value argument is a long integer. The letter x prints out hexadecimal numbers using the upper case letters *A-F* rather than *a-* f as would be printed when using x. When the alternate format is specified, a leading zero will be supplied for the octal format, and a leading 0x or 0X for the HEX format.

s Print a string - the value argument is assumed to be a character pointer. At most n characters from the string will be printed, in a field m characters wide.

c The argument is assumed to be a single character and is printed literally.

Any other characters used as conversion specifications will be printed. Thus % will produce a single percent sign.

The vprintf() function is similar to printf() but takes a variable argument list pointer rather than a list of arguments. See the description of va_start() for more information on variable argument lists. An example of using vprintf() is given below.

Example

```
printf(" = %4d%" 23)
       yields 'Total =   23%'

printf(" is %lx", size)
       where size is a long, prints size
       as hexadecimal.

printf(" = %.8s" ""
       yields 'Name = a1234567'

printf("%*d" 3, 4)
       yields 'xx  4'


/* vprintf example */

#include        <stdio.h>

int
error (char * s, ...)
{
    va_list ap;

    va_start(ap, s);
    printf(": ";
    vprintf(s, ap);
    putchar('\n');
    va_end(ap);
}

void
main (void)
{
    int i;

    i = 3;
    error(" 1 2 %d" i);
}
```

See Also

```
sprintf()
```

Return Value

The printf() and vprintf() functions return the number of characters written to stdout.

PUTCHAR

Synopsis

```
#include <stdio.h>

int putchar (int c)
```

Description

The `putchar()` function calls `putch()` to print one character to stdout, and is defined in stdio.h.

Example

```
#include <stdio.h>

char * x = " is a string"

void
main (void)
{
    char * cp;

    cp = x;
    while(*x)
        putchar(*x++);
    putchar('\n');
}
```

See Also

`putc()`, `getc()`, `freopen()`, `fclose()`

Return Value

The character passed as argument, or EOF if an error occurred.

PUTS

Synopsis

```
#include <stdio.h>

int puts (const char * s)
```

Description

The `puts()` function writes the string s to the stdout stream, appending a newline. The null character terminating the string is not copied.

Example

```
#include <stdio.h>

void
main (void)
{
    puts(", world!";
}
```

See Also

`fputs()`, `gets()`, `freopen()`, `fclose()`

Return Value

EOF is returned on error; zero otherwise.

QSORT

Synopsis

```
#include <stdlib.h>

void qsort (void * base, size_t nel, size_t width,
int (*func)(const void *, const void *))
```

Description

The `qsort()` function is an implementation of the quicksort algorithm. It sorts an array of `nel` items, each of length `width` bytes, located contiguously in memory at `base`. The argument `func` is a pointer to a function used by `qsort()` to compare items. It calls `func` with pointers to two items to be compared. If the first item is considered to be greater than, equal to or less than the second then `func` should return a value greater than zero, equal to zero or less than zero respectively.

Example

```
#include <stdio.h>
#include <stdlib.h>

int array[] = {
    567, 23, 456, 1024, 17, 567, 66
};

int
sortem (const void * p1, const void * p2)
{
    return *(int *)p1 - *(int *)p2;
}

void
main (void)
{
    register int i;

    qsort(aray, sizeof array/sizeof array[0],
        sizeof array[0], sortem);
    for(i = 0 ; i != sizeof array/sizeof array[0] ; i++)
        printf("d\t" array[i]);
    putchar('\n');
}
```

Note

The function parameter must be a pointer to a function of type similar to:

```
int func (const void *, const void *)
```

i.e. it must accept two `const void *` parameters, and must be prototyped.

RAND

Synopsis

```
#include <stdlib.h>

int rand (void)
```

Description

The `rand()` function is a pseudo-random number generator. It returns an integer in the range 0 to 32767, which changes in a pseudo-random fashion on each call. The algorithm will produce a deterministic sequence if started from the same point. The starting point is set using the `srand()` call. The example shows use of the `time()` function to generate a different starting point for the sequence each time.

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

void
main (void)
{
    time_t toc;
    int i;

    time(&toc);
    srand((int)toc);
    for(i = 0 ; i != 10 ; i++)
        printf("d\t" rand());
    putchar('\n');
}
```

See Also

```
srand()
```

Note

The example will require the user to provide the `time()` routine as one cannot be supplied with the compiler. See `time()` for more detail.

ROUND

Synopsis

```
#include <math.h>

double round (double x)
```

Description

The `round` function round the argument to the nearest integer value, but in floating-point format. Values midway between integer values are rounded up.

Example

```
#include <math.h>

void
main (void)
{
    double input, rounded;
    input = 1234.5678;
    rounded = round(input);
}
```

See Also

```
trunc()
```

SETJMP

Synopsis

```
#include <setjmp.h>

int setjmp (jmp_buf buf)
```

Description

The setjmp() function is used with longjmp() for non-local goto's. See longjmp() for further information.

Example

```
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>

jmp_buf jb;

void
inner (void)
{
    longjmp(jb, 5);
}

void
main (void)
{
    int i;

    if(i = setjmp(jb)) {
        printf(" returned %d\n" i);
        exit(0);
    }
    printf(" returned 0 - good\n";
    printf(" inner...\n";
    inner();
    printf(" returned - bad!\n";
}
```

See Also

```
longjmp()
```

Return Value

The `setjmp()` function returns zero after the real call, and non-zero if it apparently returns after a call to `longjmp()`.

SIN

Synopsis

```
#include <math.h>

double sin (double f)
```

Description

This function returns the sine function of its argument.

Example

```
#include <math.h>
#include <stdio.h>

#define C 3.141592/180.0

void
main (void)
{
    double i;

    for(i = 0 ; i <= 180.0 ; i += 10)
        printf("(%3.0f) = %f\n" i, sin(i*C));
        printf("(%3.0f) = %f\n" i, cos(i*C));
}
```

See Also

`cos(), tan(), asin(), acos(), atan(), atan2()`

Return Value

Sine vale of `f`.

SLEEP

Synopsis

```
#include <htc.h>

SLEEP();
```

Description

This macro is used to put the device into a low-power standby mode.

Example

```
#include <htc.h>
extern void init(void);

void
main (void)
{
    init(); /* enable peripherals/interrupts */

    while(1)
        SLEEP();  /* save power while nothing happening */
}
```

SPRINTF, VSPRINTF

Synopsis

```
#include <stdio.h>

int sprintf (char * buf, const char * fmt, ...)

#include <stdio.h>
#include <stdarg.h>

int vsprintf (char * buf, const char * fmt, va_list ap)
```

Description

The sprintf() function operates in a similar fashion to printf(), except that instead of placing the converted output on the stdout stream, the characters are placed in the buffer at buf. The resultant string will be null terminated, and the number of characters in the buffer will be returned.

The vsprintf() function is similar to sprintf() but takes a variable argument list pointer rather than a list of arguments. See the description of va_start() for more information on variable argument lists.

See Also

printf(), sscanf()

Return Value

Both these routines return the number of characters placed into the buffer.

SQRT

Synopsis

```
#include <math.h>

double sqrt (double f)
```

Description

The function sqrt(), implements a square root routine using Newton's approximation.

Example

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    double i;

    for(i = 0 ; i <= 20.0 ; i += 1.0)
        printf(" root of %.1f = %f\n" i, sqrt(i));
}
```

See Also

```
exp()
```

Return Value

Returns the value of the square root.

Note

A domain error occurs if the argument is negative.

SRAND

Synopsis

```
#include <stdlib.h>
```

```
void srand (unsigned int seed)
```

Description

The srand() function initializes the random number generator accessed by rand() with the given seed. This provides a mechanism for varying the starting point of the pseudo-random sequence yielded by rand().

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

void
main (void)
{
    time_t toc;
    int i;

    time(&toc);
    srand((int)toc);
    for(i = 0 ; i != 10 ; i++)
        printf("d\t" rand());
    putchar('\n');
}
```

See Also

```
rand()
```

SSCANF, VSSCANF

Synopsis

```
#include <stdio.h>

int sscanf (const char * buf, const char * fmt, ...)

#include <stdio.h>
#include <stdarg.h>

int vsscanf (const char * buf, const char * fmt, va_list ap)
```

Description

The `sscanf()` function operates in a similar manner to `scanf()`, except that instead of the conversions being taken from stdin, they are taken from the string at `buf`.

The `vsscanf()` function takes an argument pointer rather than a list of arguments. See the description of `va_start()` for more information on variable argument lists.

See Also

`scanf()`, `fscanf()`, `sprintf()`

Return Value

Returns the value of EOF if an input failure occurs, else returns the number of input items.

STRCAT

Synopsis

```
#include <string.h>

char * strcat (char * s1, const char * s2)
```

Description

This function appends (concatenates) string `s2` to the end of string `s1`. The result will be null terminated. The argument `s1` must point to a character array big enough to hold the resultant string.

Example

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    char buffer[256];
    char * s1, * s2;

    strcpy(buffer, " of line";
    s1 = buffer;
    s2 = "... end of line"
    strcat(s1, s2);
    printf(" = %d\n" strlen(buffer));
    printf(" = \"s\"n" buffer);
}
```

See Also

strcpy(), strcmp(), strncat(), strlen()

Return Value

The value of s1 is returned.

STRCHR, STRICHR

Synopsis

```
#include <string.h>

char * strchr (const char * s, int c)
char * strichr (const char * s, int c)
```

Description

The strchr() function searches the string s for an occurrence of the character c. If one is found, a pointer to that character is returned, otherwise NULL is returned.

The strichr() function is the case-insensitive version of this function.

Example

```
#include <strings.h>
#include <stdio.h>

void
main (void)
{
    static char temp[] = " it is..."
    char c = 's';

    if(strchr(temp, c))
        printf(" %c was found in string\n" c);
    else
        printf(" character was found in string";
}
```

See Also

strrchr(), strlen(), strcmp()

Return Value

A pointer to the first match found, or NULL if the character does not exist in the string.

Note

Although the function takes an integer argument for the character, only the lower 8 bits of the value are used.

STRCMP, STRICMP

Synopsis

```
#include <string.h>

int strcmp (const char * s1, const char * s2)
int stricmp (const char * s1, const char * s2)
```

Description

The `strcmp()` function compares its two, null terminated, string arguments and returns a signed integer to indicate whether `s1` is less than, equal to or greater than `s2`. The comparison is done with the standard collating sequence, which is that of the ASCII character set.

The `stricmp()` function is the case-insensitive version of this function.

Example

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    int i;

    if((i = strcmp("" "") < 0)
        printf(" is less than ABc\n";
    else if(i > 0)
        printf(" is greater than ABc\n";
    else
        printf(" is equal to ABc\n";
}
```

See Also

`strlen()`, `strncmp()`, `strcpy()`, `strcat()`

Return Value

A signed integer less than, equal to or greater than zero.

Note

Other C implementations may use a different collating sequence; the return value is negative, zero or positive, i.e. do not test explicitly for negative one (-1) or one (1).

STRCPY

Synopsis

```
#include <string.h>

char * strcpy (char * s1, const char * s2)
```

Description

This function copies a null terminated string `s2` to a character array pointed to by `s1`. The destination array must be large enough to hold the entire string, including the null terminator.

Example

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    char buffer[256];
    char * s1, * s2;

    strcpy(buffer, " of line";
    s1 = buffer;
    s2 = "... end of line"
    strcat(s1, s2);
    printf(" = %d\n" strlen(buffer));
    printf(" = \"s\"n" buffer);
}
```

See Also

```
strncpy(), strlen(), strcat(), strlen()
```

Return Value

The destination buffer pointer `s1` is returned.

STRCSPN

Synopsis

```
#include <string.h>

size_t strcspn (const char * s1, const char * s2)
```

Description

The `strcspn()` function returns the length of the initial segment of the string pointed to by `s1` which consists of characters NOT from the string pointed to by `s2`.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    static char set[] = ""

    printf("d\n" strcspn( "" set));
    printf("d\n" strcspn( "" set));
    printf("d\n" strcspn( "" set));
}
```

See Also

```
strspn()
```

Return Value

Returns the length of the segment.

STRLEN

Synopsis

```
#include <string.h>

size_t strlen (const char * s)
```

Description

The `strlen()` function returns the number of characters in the string `s`, not including the null terminator.

Example

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    char buffer[256];
    char * s1, * s2;

    strcpy(buffer, " of line";
    s1 = buffer;
    s2 = "... end of line"
    strcat(s1, s2);
    printf(" = %d\n" strlen(buffer));
    printf(" = \"s\"n" buffer);
}
```

Return Value

The number of characters preceding the null terminator.

STRNCAT

Synopsis

```
#include <string.h>

char * strncat (char * s1, const char * s2, size_t n)
```

Description

This function appends (concatenates) string `s2` to the end of string `s1`. At most `n` characters will be copied, and the result will be null terminated. `s1` must point to a character array big enough to hold the resultant string.

Example

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    char buffer[256];
    char * s1, * s2;

    strcpy(buffer, " of line";
    s1 = buffer;
    s2 = "... end of line"
    strncat(s1, s2, 5);
    printf(" = %d\n" strlen(buffer));
    printf(" = \"s\"n" buffer);
}
```

See Also

`strcpy()`, `strcmp()`, `strcat()`, `strlen()`

Return Value

The value of `s1` is returned.

STRNCMP, STRNICMP

Synopsis

```
#include <string.h>

int strncmp (const char * s1, const char * s2, size_t n)
int strnicmp (const char * s1, const char * s2, size_t n)
```

Description

The `strncmp()` function compares its two, null terminated, string arguments, up to a maximum of `n` characters, and returns a signed integer to indicate whether `s1` is less than, equal to or greater than `s2`. The comparison is done with the standard collating sequence, which is that of the ASCII character set.

The `strnicmp()` function is the case-insensitive version of this function.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    int i;

    i = strncmp("" ""6);
    if(i == 0)
        printf(" strings are equal\n";
    else if(i > 0)
        printf(" 2 less than string 1\n";
    else
        printf(" 2 is greater than string 1\n";
}
```

See Also

strlen(), strcmp(), strcpy(), strcat()

Return Value

A signed integer less than, equal to or greater than zero.

Note

Other C implementations may use a different collating sequence; the return value is negative, zero or positive, i.e. do not test explicitly for negative one (-1) or one (1).

STRNCPY

Synopsis

```
#include <string.h>
```

```
char * strncpy (char * s1, const char * s2, size_t n)
```

Description

This function copies a null terminated string s2 to a character array pointed to by s1. At most n characters are copied. If string s2 is longer than n then the destination string will not be null terminated. The destination array must be large enough to hold the entire string, including the null terminator.

Example

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    char buffer[256];
    char * s1, * s2;

    strncpy(buffer, " of line" 6);
    s1 = buffer;
    s2 = "... end of line"
    strcat(s1, s2);
    printf(" = %d\n" strlen(buffer));
    printf(" = \"s\"n" buffer);
}
```

See Also

strcpy(), strcat(), strlen(), strcmp()

Return Value

The destination buffer pointer s1 is returned.

STRPBRK

Synopsis

```
#include <string.h>

char * strpbrk (const char * s1, const char * s2)
```

Description

The `strpbrk()` function returns a pointer to the first occurrence in string `s1` of any character from string `s2`, or a null pointer if no character from `s2` exists in `s1`.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    char * str = " is a string."

    while(str != NULL) {
        printf( "s\n" str );
        str = strpbrk( str+1, "");
    }
}
```

Return Value

Pointer to the first matching character, or `NULL` if no character found.

STRRCHR, STRRICHR

Synopsis

```
#include <string.h>

char * strrchr (char * s, int c)
char * strrichr (char * s, int c)
```

Description

The `strrchr()` function is similar to the `strchr()` function, but searches from the end of the string rather than the beginning, i.e. it locates the *last* occurrence of the character `c` in the null terminated string `s`. If successful it returns a pointer to that occurrence, otherwise it returns NULL.

The `strrichr()` function is the case-insensitive version of this function.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    char * str = " is a string."

    while(str != NULL) {
        printf( "s\n" str );
        str = strrchr( str+1, 's');
    }
}
```

See Also

`strchr(), strlen(), strcmp(), strcpy(), strcat()`

Return Value

A pointer to the character, or NULL if none is found.

STRSPN

Synopsis

```
#include <string.h>

size_t strspn (const char * s1, const char * s2)
```

Description

The `strspn()` function returns the length of the initial segment of the string pointed to by `s1` which consists entirely of characters from the string pointed to by `s2`.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    printf("d\n" strspn(" is a string" ""));
    printf("d\n" strspn(" is a string" ""));
}
```

See Also

`strcspn()`

Return Value

The length of the segment.

STRSTR, STRISTR

Synopsis

```
#include <string.h>

char * strstr (const char * s1, const char * s2)
char * stristr (const char * s1, const char * s2)
```

Description

The `strstr()` function locates the first occurrence of the sequence of characters in the string pointed to by `s2` in the string pointed to by `s1`.

The `stristr()` routine is the case-insensitive version of this function.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    printf("d\n" strstr(" is a string" "");
}
```

Return Value

Pointer to the located string or a null pointer if the string was not found.

STRTOD

Synopsis

```
#include <stdlib.h>

double strtok (const char * s, const char ** res)
```

Description

Parse the string `s` converting it to a double floating point type. This function converts the first occurrence of a substring of the input that is made up of characters of the expected form after skipping leading white-space characters. If `res` is not `NULL`, it will be made to point to the first character after the converted sub-string.

Example

```
#include <stdio.h>
#include <strlib.h>

void
main (void)
{
    char buf[] = "35.7  23.27 "
    char * end;
    double in1, in2;

    in1 = strtod(buf, &end);
    in2 = strtod(end, NULL);
    printf(" comps: %f, %f\n" in1, in2);
}
```

See Also

atof()

Return Value

Returns a double representing the floating-point value of the converted input string.

STRTOL

Synopsis

```
#include <stdlib.h>

double strtol (const char * s, const char ** res, int base)
```

Description

Parse the string s converting it to a long integer type. This function converts the first occurrence of a substring of the input that is made up of characters of the expected form after skipping leading white-space characters. The radix of the input is determined from base. If this is zero, then the radix defaults to base 10. If res is not NULL, it will be made to point to the first character after the converted sub-string.

Example

```
#include <stdio.h>
#include <strlib.h>

void
main (void)
{
    char buf[] = "0X299 0x792 "
    char * end;
    long in1, in2;

    in1 = strtol(buf, &end, 16);
    in2 = strtol(end, NULL, 16);
    printf(" (decimal): %ld, %ld\n" in1, in2);
}
```

See Also

strtod()

Return Value

Returns a long int representing the value of the converted input string using the specified base.

STRTOK

Synopsis

```
#include <string.h>

char * strtok (char * s1, const char * s2)
```

Description

A number of calls to strtok() breaks the string s1 (which consists of a sequence of zero or more text tokens separated by one or more characters from the separator string s2) into its separate tokens.

The first call must have the string s1. This call returns a pointer to the first character of the first token, or NULL if no tokens were found. The inter-token separator character is overwritten by a null character, which terminates the current token.

For subsequent calls to strtok(), s1 should be set to a NULL pointer. These calls start searching from the end of the last token found, and again return a pointer to the first character of the next token, or NULL if no further tokens were found.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    char * ptr;
    char buf[] = " is a string of words."
    char * sep_tok = ",?! "

    ptr = strtok(buf, sep_tok);
    while(ptr != NULL) {
        printf("s\n" ptr);
        ptr = strtok(NULL, sep_tok);
    }
}
```

Return Value

Returns a pointer to the first character of a token, or a null pointer if no token was found.

Note

The separator string s2 may be different from call to call.

TAN

Synopsis

```
#include <math.h>

double tan (double f)
```

Description

The tan() function calculates the tangent of f.

Example

```
#include <math.h>
#include <stdio.h>

#define C 3.141592/180.0

void
main (void)
{
    double i;

    for(i = 0 ; i <= 180.0 ; i += 10)
        printf("(%3.0f) = %f\n" i, tan(i*C));
}
```

See Also

sin(), cos(), asin(), acos(), atan(), atan2()

Return Value

The tangent of f.

TIME

Synopsis

```
#include <time.h>

time_t time (time_t * t)
```

Description

This function is not provided as it is dependant on the target system supplying the current time. This function will be user implemented. When implemented, this function should return the current time in seconds since 00:00:00 on Jan 1, 1970. If the argument t is not equal to NULL, the same value is stored into the object pointed to by t.

Example

```
#include <stdio.h>
#include <time.h>

void
main (void)
{
    time_t clock;

    time(&clock);
    printf("s" ctime(&clock));
}
```

See Also

ctime(), gmtime(), localtime(), asctime()

Return Value

This routine when implemented will return the current time in seconds since 00:00:00 on Jan 1, 1970.

Note

The `time()` routine is not supplied, if required the user will have to implement this routine to the specifications outlined above.

TOLOWER, TOUPPER, TOASCII

Synopsis

```
#include <ctype.h>

char toupper (int c)
char tolower (int c)
char toascii (int c)
```

Description

The `toupper()` function converts its lower case alphabetic argument to upper case, the `tolower()` routine performs the reverse conversion and the `toascii()` macro returns a result that is guaranteed in the range 0-0177. The functions `toupper()` and `tolower()` return their arguments if it is not an alphabetic character.

Example

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

void
main (void)
{
    char * array1 = ""
    int i;

    for(i=0;i < strlen(array1); ++i) {
        printf("c" tolower(array1[i]));
    }
    printf("n";
}
```

See Also

`islower()`, `isupper()`, `isascii()`, et. al.

TRUNC

Synopsis

```
#include <math.h>

double trunc (double x)
```

Description

The `trunc` function rounds the argument to the nearest integer value, in floating-point format, that is not larger in magnitude than the argument.

Example

```
#include <math.h>

void
main (void)
{
    double input, rounded;
    input = 1234.5678;
    rounded = trunc(input);
}
```

See Also

```
round()
```

UDIV

Synopsis

```
#include <stdlib.h>

int udiv (unsigned num, unsigned demon)
```

Description

The udiv() function calculate the quotient and remainder of the division of number and denom, storing the results into a udiv_t structure which is returned.

Example

```
#include <stdlib.h>

void
main (void)
{
    udiv_t result;
    unsigned  num = 1234, den = 7;

    result = udiv(num, den);
}
```

See Also

```
uldiv(), div(), ldiv()
```

Return Value

Returns the quotient and remainder as a udiv_t structure.

ULDIV

Synopsis

```
#include <stdlib.h>

int uldiv (unsigned long num, unsigned long demon)
```

Description

The uldiv() function calculate the quotient and remainder of the division of number and denom, storing the results into a uldiv_t structure which is returned.

Example

```
#include <stdlib.h>

void
main (void)
{
    uldiv_t result;
    unsigned long num = 1234, den = 7;

    result = uldiv(num, den);
}
```

See Also

`ldiv(), udiv(), div()`

Return Value

Returns the quotient and remainder as a `uldiv_t` structure.

UTOA

Synopsis

```
#include <stdlib.h>

char * utoa (char * buf, unsigned val, int base)
```

Description

The function `itoa()` converts the unsigned contents of `val` into a string which is stored into `buf`. The conversion is performed according to the radix specified in `base`. `buf` is assumed to reference a buffer which has sufficient space allocated to it.

Example

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    char buf[10];
    utoi(buf, 1234, 16);
    printf(" buffer holds %s\n" buf);
}
```

See Also

`strtol(), itoa(), ltoa(), ultoa()`

Return Value

This routine returns a copy of the buffer into which the result is written.

VA_START, VA_ARG, VA_END

Synopsis

```
#include <stdarg.h>

void va_start (va_list ap, parmN)
type  va_arg  (ap, type)
void va_end  (va_list ap)
```

Description

These macros are provided to give access in a portable way to parameters to a function represented in a prototype by the ellipsis symbol (. . .), where type and number of arguments supplied to the function are not known at compile time.

The right most parameter to the function (shown as *parmN*) plays an important role in these macros, as it is the starting point for access to further parameters. In a function taking variable numbers of arguments, a variable of type `va_list` should be declared, then the macro `va_start()` invoked with that variable and the name of *parmN*. This will initialize the variable to allow subsequent calls of the macro `va_arg()` to access successive parameters.

Each call to `va_arg()` requires two arguments; the variable previously defined and a type name which is the type that the next parameter is expected to be. Note that any arguments thus accessed will have been widened by the default conventions to `int`, `unsigned int` or `double`. For example if a character argument has been passed, it should be accessed by `va_arg(ap, int)` since the `char` will have been widened to `int`.

An example is given below of a function taking one integer parameter, followed by a number of other parameters. In this example the function expects the subsequent parameters to be pointers to `char`, but note that the compiler is not aware of this, and it is the programmers responsibility to ensure that correct arguments are supplied.

Example

```
#include <stdio.h>
#include <stdarg.h>

void
pf (int a, ...)
{
    va_list ap;

    va_start(ap, a);
    while(a--)
        puts(va_arg(ap, char *));
    va_end(ap);
}

void
main (void)
{
    pf(3, " 1" " 2" " 3";
}
```

XTOI

Synopsis

```
#include <stdlib.h>

unsigned xtoi (const char * s)
```

Description

The xtoi() function scans the character string passed to it, skipping leading blanks reading an optional sign, and converts an ASCII representation of a hexadecimal number to an integer.

Example

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    char buf[80];
    int i;

    gets(buf);
    i = xtoi(buf);
    printf(" %s: converted to %x\n" buf, i);
}
```

See Also

```
atoi()
```

Return Value

An unsigned integer. If no number is found in the string, zero will be returned.

NOTES:

Chapter 8. Error and Warning Messages

This chapter lists most error, warning and advisory messages from all HI-TECH C compilers, with an explanation of each message. Most messages have been assigned a unique number which appears in brackets before each message description, and which is also printed by the compiler when the message is issued. The messages shown here are sorted by their number. Un-numbered messages appear toward the end and are sorted alphabetically.

The name of the application(s) that could have produced the messages are listed in brackets opposite the error message. In some cases examples of code or options that could trigger the error are given. The use of * in the error message is used to represent a string that the compiler will substitute that is specific to that particular error.

Note that one problem in your C or assembler source code may trigger more than one error message. You should attempt to resolve errors or warnings in the order in which they are produced.

(1) too many errors (*)   *(all applications)*

The executing compiler application has encountered too many errors and will exit immediately. Other uncompiled source files will be processed, but the compiler applications that would normally be executed in due course will not be run. The number of errors that can be accepted can be controlled using the `--ERRORS` option, See **Section 2.7.28 "--ERRORS: Maximum Number of Errors"**.

(2) error/warning (*) generated, but no description available   *(all applications)*

The executing compiler application has emitted a message (advisory/warning/error), but there is no description available in the message description file (MDF) to print. This may be because the MDF is out of date, or the message issue has not been translated into the selected language.

(3) malformed error information on line *, in file *   (*all applications*)

The compiler has attempted to load the messages for the selected language, but the message description file (MDF) was corrupted and could not be read correctly.

(100) unterminated #if[n][def] block from line *   *(Preprocessor)*

A `#if` or similar block was not terminated with a matching `#endif`, e.g.:

```
#if INPUT        /* error flagged here */
void main(void)
{
  run();
}                /* no #endif was found in this module */
```

(101) #* may not follow #else *(Preprocessor)*

A `#else` or `#elif` has been used in the same conditional block as a `#else`. These can only follow a `#if`, e.g.:

```
#ifdef FOO
  result = foo;
#else
  result = bar;
#elif defined(NEXT)   /* the #else above terminated the #if */
  result = next(0);
#endif
```

(102) #* must be in an #if *(Preprocessor)*

The `#elif`, `#else` or `#endif` directive must be preceded by a matching `#if` line. If there is an apparently corresponding `#if` line, check for things like extra `#endif`'s, or improperly terminated comments, e.g.:

```
#ifdef FOO
  result = foo;
#endif
  result = bar;
#elif defined(NEXT)   /* the #endif above terminated the #if */
  result = next(0);
#endif
```

(103) #error: * *(Preprocessor)*

This is a programmer generated error; there is a directive causing a deliberate error. This is normally used to check compile time defines etc. Remove the directive to remove the error, but first check as to why the directive is there.

(104) preprocessor #assert failure *(Preprocessor)*

The argument to a preprocessor `#assert` directive has evaluated to zero. This is a programmer induced error.

```
#assert SIZE == 4  /* size should never be 4 */
```

(105) no #asm before #endasm *(Preprocessor)*

A `#endasm` operator has been encountered, but there was no previous matching `#asm`, e.g.:

```
void cleardog(void)
{
  clrwdt
#endasm  /* in-line assembler ends here,
          only where did it begin? */
}
```

(106) nested #asm directives *(Preprocessor)*

It is not legal to nest `#asm` directives. Check for a missing or misspelled `#endasm` directive, e.g.:

```
#asm
    MOVE   r0, #0aah
#asm       ; previous #asm must be closed before opening another
    SLEEP
#endasm
```

(107) illegal # directive "*"   *(Preprocessor, Parser)*

The compiler does not understand the # directive. It is probably a misspelling of a
preprocessor # directive, e.g.:

```
#indef DEBUG  /* oops -- that should be #undef DEBUG */
```

(108) #if[n][def] without an argument   *(Preprocessor)*

The preprocessor directives #if, #ifdef and #ifndef must have an argument. The
argument to #if should be an expression, while the argument to #ifdef or #ifndef
should be a single name, e.g.:

```
#if           /* oops -- no argument to check */
  output = 10;
#else
  output = 20;
#endif
```

(109) #include syntax error   *(Preprocessor)*

The syntax of the filename argument to #include is invalid. The argument to
#include must be a valid file name, either enclosed in double quotes " " or angle
brackets < >. Spaces should not be included, and the closing quote or bracket must
be present. There should be nothing else on the line other than comments, e.g.:

```
#include stdio.h  /* oops -- should be: #include <stdio.h> */
```

(110) too many file arguments; usage: cpp [input [output]]
(Preprocessor)

CPP should be invoked with at most two file arguments. Contact HI-TECH Support if
the preprocessor is being executed by a compiler driver.

(111) redefining preprocessor macro "*"   *(Preprocessor)*

The macro specified is being redefined, to something different to the original definition.
If you want to deliberately redefine a macro, use #undef first to remove the original
definition, e.g.:

```
#define ONE 1
/* elsewhere: */
/* Is this correct? It will overwrite the first definition. */
#define ONE one
```

(112) #define syntax error   *(Preprocessor)*

A macro definition has a syntax error. This could be due to a macro or formal parameter
name that does not start with a letter or a missing *closing parenthesis*,), e.g.:

```
#define FOO(a, 2b)  bar(a, 2b)  /* 2b is not to be! */
```

(113) unterminated string in preprocessor macro body   *(Preprocessor, Assembler)*

A macro definition contains a string that lacks a closing quote.

(114) illegal #undef argument   *(Preprocessor)*

The argument to #undef must be a valid name. It must start with a letter, e.g.:

```
#undef 6YYY  /* this isn't a valid symbol name */
```

(115) recursive preprocessor macro definition of "*" defined by "*" *(Preprocessor)*

The named macro has been defined in such a manner that expanding it causes a recursive expansion of itself!

(116) end of file within preprocessor macro argument from line * *(Preprocessor)*

A macro argument has not been terminated. This probably means the closing parenthesis has been omitted from a macro invocation. The line number given is the line where the macro argument started, e.g.:

```
#define FUNC(a, b) func(a+b)
FUNC(5, 6;    /* oops -- where is the closing bracket? */
```

(117) misplaced constant in #if *(Preprocessor)*

A constant in a `#if` expression should only occur in syntactically correct places. This error is most probably caused by omission of an operator, e.g.:

```
#if FOO BAR  /* oops -- did you mean: #if FOO == BAR ? */
```

(118) stack overflow processing #if expression *(Preprocessor)*

The preprocessor filled up its expression evaluation stack in a `#if` expression. Simplify the expression — it probably contains too many parenthesized subexpressions.

(119) invalid expression in #if line *(Preprocessor)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(120) operator "*" in incorrect context *(Preprocessor)*

An operator has been encountered in a `#if` expression that is incorrectly placed, e.g. two binary operators are not separated by a value, e.g.:

```
#if FOO * % BAR == 4  /* what is "* %" ? */
  #define BIG
#endif
```

(121) expression stack overflow at operator "*" *(Preprocessor)*

Expressions in `#if` lines are evaluated using a stack with a size of 128. It is possible for very complex expressions to overflow this. Simplify the expression.

(122) unbalanced parenthesis at operator "*" *(Preprocessor)*

The evaluation of a `#if` expression found mismatched parentheses. Check the expression for correct parenthesization, e.g.:

```
#if ((A) + (B)  /* oops -- a missing ), I think */
  #define ADDED
#endif
```

(123) misplaced "?" or ":"; previous operator is "*" *(Preprocessor)*

A colon operator has been encountered in a `#if` expression that does not match up with a corresponding `?` operator, e.g.:

```
#if XXX : YYY  /* did you mean:  #if COND ? XXX : YYY */
```

(124) illegal character "*" in #if   *(Preprocessor)*

There is a character in a `#if` expression that has no business being there. Valid characters are the letters, digits and those comprising the acceptable operators, e.g.:

```
#if YYY  /* what are these characters doing here? */
  int m;
#endif
```

(125) illegal character (* decimal) in #if   *(Preprocessor)*

There is a non-printable character in a `#if` expression that has no business being there. Valid characters are the letters, digits and those comprising the acceptable operators, e.g.:

```
#if ^S YYY  /* what is this control characters doing here? */
  int m;
#endif
```

(126) strings can't be used in #if   *(Preprocessor)*

The preprocessor does not allow the use of strings in `#if` expressions, e.g.:

```
/* no string operations allowed by the preprocessor */
#if MESSAGE > "hello"
#define DEBUG
#endif
```

(127) bad syntax for defined() in #[el]if   *(Preprocessor)*

The `defined()` pseudo-function in a preprocessor expression requires its argument to be a single name. The name must start with a letter and should be enclosed in parentheses, e.g.:

```
/* oops -- defined expects a name, not an expression */
#if defined(a&b)
  input = read();
#endif
```

(128) illegal operator in #if   *(Preprocessor)*

A `#if` expression has an illegal operator. Check for correct syntax, e.g.:

```
#if FOO = 6  /* oops -- should that be: #if FOO == 5 ? */
```

(129) unexpected "\" in #if   *(Preprocessor)*

The *backslash* is incorrect in the `#if` statement, e.g.:

```
#if FOO == \34
  #define BIG
#endif
```

(130) unknown type "*" in #[el]if sizeof()   *(Preprocessor)*

An unknown type was used in a preprocessor `sizeof()`. The preprocessor can only evaluate `sizeof()` with basic types, or pointers to basic types, e.g.:

```
#if sizeof(unt) == 2  /* should be: #if sizeof(int) == 2 */
  i = 0xFFFF;
#endif
```

(131) illegal type combination in #[el]if sizeof()   *(Preprocessor)*

The preprocessor found an illegal type combination in the argument to `sizeof()` in a
`#if` expression, e.g.

```
/* To sign, or not to sign, that is the error. */
#if sizeof(signed unsigned int) == 2
  i = 0xFFFF;
#endif
```

(132) no type specified in #[el]if sizeof()   *(Preprocessor)*

Sizeof() was used in a preprocessor `#if` expression, but no type was specified. The
argument to `sizeof()` in a preprocessor expression must be a valid simple type, or
pointer to a simple type, e.g.:

```
#if sizeof()  /* oops -- size of what? */
  i = 0;
#endif
```

(133) unknown type code (0x*) in #[el]if sizeof()   *(Preprocessor)*

The preprocessor has made an internal error in evaluating a `sizeof()` expression.
Check for a malformed type specifier. This is an internal error. Contact HI-TECH
Software technical support with details.

(134) syntax error in #[el]if sizeof()   *(Preprocessor)*

The preprocessor found a syntax error in the argument to `sizeof`, in a `#if` expres-
sion. Probable causes are mismatched parentheses and similar things, e.g.:

```
#if sizeof(int == 2)  // oops - should be: #if sizeof(int) == 2
  i = 0xFFFF;
#endif
```

(135) unknown operator (*) in #if   *(Preprocessor)*

The preprocessor has tried to evaluate an expression with an operator it does not
understand. This is an internal error. Contact HI-TECH Software technical support with
details.

(137) strange character "*" after ##   *(Preprocessor)*

A character has been seen after the token catenation operator `##` that is neither a letter
nor a digit. Since the result of this operator must be a legal token, the operands must
be tokens containing only letters and digits, e.g.:

```
/* the ' character will not lead to a valid token */
#define cc(a, b) a ## 'b
```

(138) strange character (*) after ##   *(Preprocessor)*

An unprintable character has been seen after the token catenation operator `##` that is
neither a letter nor a digit. Since the result of this operator must be a legal token, the
operands must be tokens containing only letters and digits, e.g.:

```
/* the ' character will not lead to a valid token */
#define cc(a, b) a ## 'b
```

(139) end of file in comment   *(Preprocessor)*

End of file was encountered inside a comment. Check for a missing closing comment
flag, e.g.:

```
    /* Here the comment begins. I'm not sure where I end, though
}
```

(140) can't open * file "*": * *(Driver, Preprocessor, Code Generator, Assembler)*

The command file specified could not be opened for reading. Confirm the spelling and path of the file specified on the command line, e.g.:

```
picc @communds
```

should that be:

```
picc @commands
```

(141) can't open * file "*": * *(Any)*

An output file could not be created. Confirm the spelling and path of the file specified on the command line.

(144) too many nested #if blocks *(Preprocessor)*

`#if`, `#ifdef` etc. blocks may only be nested to a maximum of 32.

(146) #include filename too long *(Preprocessor)*

A filename constructed while looking for an include file has exceeded the length of an internal buffer. Since this buffer is 4096 bytes long, this is unlikely to happen.

(147) too many #include directories specified *(Preprocessor)*

A maximum of 7 directories may be specified for the preprocessor to search for include files. The number of directories specified with the driver is too great.

(148) too many arguments for preprocessor macro *(Preprocessor)*

A macro may only have up to 31 parameters, as per the C Standard.

(149) preprocessor macro work area overflow *(Preprocessor)*

The total length of a macro expansion has exceeded the size of an internal table. This table is normally 32768 bytes long. Thus any macro expansion must not expand into a total of more than 32K bytes.

(150) illegal "__" preprocessor macro "*" *(Preprocessor)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(151) too many arguments in preprocessor macro expansion *(Preprocessor)*

There were too many arguments supplied in a macro invocation. The maximum number allowed is 31.

(152) bad dp/nargs in openpar(): c = * *(Preprocessor)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(153) out of space in preprocessor macro "*" argument expansion *(Preprocessor)*

A macro argument has exceeded the length of an internal buffer. This buffer is normally 4096 bytes long.

(155) work buffer overflow concatenating "*"   *(Preprocessor)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(156) work buffer "*" overflow   *(Preprocessor)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(157) can't allocate * bytes of memory   *(Code Generator, Assembler)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(158) invalid disable in preprocessor macro "*"   *(Preprocessor)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(159) too many calls to unget()   *(Preprocessor)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(161) control line "*" within preprocessor macro expansion
                    (Preprocessor)

A preprocessor control line (one starting with a #) has been encountered while expanding a macro. This should not happen.

(162) #warning: *   *(Preprocessor, Driver)*

This warning is either the result of user-defined `#warning` preprocessor directive or the driver encountered a problem reading the map file. If the latter then please HI-TECH Software technical support with details

(163) unexpected text in control line ignored   *(Preprocessor)*

This warning occurs when extra characters appear on the end of a control line, e.g. The extra text will be ignored, but a warning is issued. It is preferable (and in accordance with Standard C) to enclose the text as a comment, e.g.:

```
#if defined(END)
  #define NEXT
#endif END      /* END would be better in a comment here */
```

(164) #include filename "*" was converted to lower case   *(Preprocessor)*

The `#include` file name had to be converted to lowercase before it could be opened, e.g.:

```
#include <STDIO.H>  /* oops -- should be: #include <stdio.h> */
```

(165) #include filename "*" does not match actual name (check
                    upper/lower case)   *(Preprocessor)*

In Windows versions this means the file to be included actually exists and is spelt the same way as the `#include` filename, however the case of each does not exactly match. For example, specifying `#include "code.c"` will include `Code.c` if it is found. In Linux versions this warning could occur if the file wasn't found.

(166) too few values specified with option "*" *(Preprocessor)*

The list of values to the preprocessor (CPP) `-S` option is incomplete. This should not happen if the preprocessor is being invoked by the compiler driver. The values passes to this option represent the sizes of `char` , `short` , `int` , `long` , `float` and `double` types.

(167) too many values specified with -S option; "*" unused *(Preprocessor)*

There were too many values supplied to the -S preprocessor option. See message 166.

(168) unknown option "*" *(Any)*

This option given to the component which caused the error is not recognized.

(169) strange character (*) after ## *(Preprocessor)*

There is an unexpected character after `#`.

(170) symbol "*" in undef was never defined *(Preprocessor)*

The symbol supplied as argument to `#undef` was not already defined. This warning may be disabled with some compilers. This warning can be avoided with code like:

```
#ifdef SYM
  #undef SYM  /* only undefine if defined */
#endif
```

(171) wrong number of preprocessor macro arguments for "*" (* instead of *) *(Preprocessor)*

A macro has been invoked with the wrong number of arguments, e.g.:

```
#define ADD(a, b) (a+b)
ADD(1, 2, 3)        /* oops -- only two arguments required */
```

(172) formal parameter expected after # *(Preprocessor)*

The stringization operator # (not to be confused with the leading # used for preprocessor control lines) must be followed by a formal macro parameter, e.g.:

```
#define str(x) #y  /* oops -- did you mean x instead of y? */
```

If you need to stringize a token, you will need to define a special macro to do it, e.g.

```
#define __mkstr__(x) #x
```

then use `__mkstr__(token)` wherever you need to convert a token into a string.

(173) undefined symbol "*" in #if, 0 used *(Preprocessor)*

A symbol on a `#if` expression was not a defined preprocessor macro. For the purposes of this expression, its value has been taken as zero. This warning may be disabled with some compilers. Example:

```
#if FOO+BAR    /* e.g. FOO was never #defined */
  #define GOOD
#endif
```

(174) multi-byte constant "*" isn't portable *(Preprocessor)*

Multi-byte constants are not portable, and in fact will be rejected by later passes of the compiler, e.g.:

```
#if CHAR == 'ab'
  #define MULTI
#endif
```

(175) division by zero in #if; zero result assumed *(Preprocessor)*

Inside a `#if` expression, there is a division by zero which has been treated as yielding zero, e.g.:

```
#if foo/0  /* divide by 0: was this what you were intending? */
  int a;
#endif
```

(176) missing newline *(Preprocessor)*

A new line is missing at the end of the line. Each line, including the last line, must have a new line at the end. This problem is normally introduced by editors.

(177) symbol "*" in -U option was never defined *(Preprocessor)*

A macro name specified in a `-U` option to the preprocessor was not initially defined, and thus cannot be undefined.

(179) nested comments *(Preprocessor)*

This warning is issued when nested comments are found. A nested comment may indicate that a previous closing comment marker is missing or malformed, e.g.:

```
output = 0;  /* a comment that was left unterminated
flag = TRUE; /* next comment:
               hey, where did this line go? */
```

(180) unterminated comment in included file *(Preprocessor)*

Comments begun inside an included file must end inside the included file.

(181) non-scalar types can't be converted to other types *(Parser)*

You can't convert a structure, union or array to another type, e.g.:

```
struct TEST test;
struct TEST * sp;
sp = test;       /* oops -- did you mean: sp = &test; ? */
```

(182) illegal conversion between types *(Parser)*

This expression implies a conversion between incompatible types, e.g. a conversion of a structure type into an integer, e.g.:

```
struct LAYOUT layout;
int i;
layout = i;          /* int cannot be converted to struct */
```

Note that even if a structure only contains an `int`, for example, it cannot be assigned to an `int` variable, and vice versa.

(183) function or function pointer required *(Parser)*

Only a function or function pointer can be the subject of a function call, e.g.:

```
int a, b, c, d;
a = b(c+d);    /* b is not a function --
                  did you mean a = b*(c+d) ? */
```

(184) calling an interrupt function is illegal *(Parser)*

A function qualified `interrupt` can't be called from other functions. It can only be called by a hardware (or software) interrupt. This is because an `interrupt` function has special function entry and exit code that is appropriate only for calling from an interrupt. An `interrupt` function can call other non-`interrupt` functions.

(185) function does not take arguments *(Parser, Code Generator)*

This function has no parameters, but it is called here with one or more arguments, e.g.:

```
int get_value(void);
void main(void)
{
  int input;
  input = get_value(6);  /* oops --
                            parameter should not be here */
}
```

(186) too many function arguments *(Parser)*

This function does not accept as many arguments as there are here.

```
void add(int a, int b);
add(5, 7, input);       /* call has too many arguments */
```

(187) too few function arguments *(Parser)*

This function requires more arguments than are provided in this call, e.g.:

```
void add(int a, int b);
add(5);                 /* this call needs more arguments */
```

(188) constant expression required *(Parser)*

In this context an expression is required that can be evaluated to a constant at compile time, e.g.:

```
int a;
switch(input) {
  case a:  /* oops!
             can't use variable as part of a case label */
    input++;
}
```

(189) illegal type for array dimension *(Parser)*

An array dimension must be either an integral type or an enumerated value.

```
int array[12.5];  /* oops -- twelve and a half elements, eh? */
```

(190) illegal type for index expression *(Parser)*

An index expression must be either integral or an enumerated value, e.g.:

```
int i, array[10];
i = array[3.5];   /* oops --
                     exactly which element do you mean? */
```

(191) cast type must be scalar or void *(Parser)*

A typecast (an abstract type declarator enclosed in parentheses) must denote a type which is either scalar (i.e. not an array or a structure) or the type `void`, e.g.:

```
lip = (long [])input;  /* oops -- maybe: lip = (long *)input */
```

(192) undefined identifier "*" *(Parser)*

This symbol has been used in the program, but has not been defined or declared. Check for spelling errors if you think it has been defined.

(193) not a variable identifier "*" *(Parser)*

This identifier is not a variable; it may be some other kind of object, e.g. a label.

(194) ")" expected *(Parser)*

A *closing parenthesis*, `)`, was expected here. This may indicate you have left out this character in an expression, or you have some other syntax error. The error is flagged on the line at which the code first starts to make no sense. This may be a statement following the incomplete expression, e.g.:

```
if(a == b  /* the closing parenthesis is missing here */
  b = 0;   /* the error is flagged here */
```

(195) expression syntax *(Parser)*

This expression is badly formed and cannot be parsed by the compiler, e.g.:

```
a /=% b;  /* oops -- maybe that should be: a /= b; */
```

(196) struct/union required *(Parser)*

A structure or union identifier is required before a *dot* `.`, e.g.:

```
int a;
a.b = 9;  /* oops -- a is not a structure */
```

(197) struct/union member expected *(Parser)*

A structure or union member name must follow a *dot* `.` or arrow (`->`).

(198) undefined struct/union "*" *(Parser)*

The specified structure or union tag is undefined, e.g.

```
struct WHAT what;  /* a definition for WHAT was never seen */
```

(199) logical type required *(Parser)*

The expression used as an operand to `if`, `while` statements or to boolean operators like `!` and `&&` must be a scalar integral type, e.g.:

```
struct FORMAT format;
if(format)             /* this operand must be a scaler type */
  format.a = 0;
```

(200) taking the address of a register variable is illegal *(Parser)*

A variable declared register may not have storage allocated for it in memory, and thus it is illegal to attempt to take the address of it by applying the & operator, e.g.:

```
int * proc(register int in)
{
  int * ip = &in;
  /* oops -- in may not have an address to take */
  return ip;
}
```

(201) taking the address of this object is illegal *(Parser)*

The expression which was the operand of the & operator is not one that denotes memory storage ("an lvalue") and therefore its address can not be defined, e.g.:

```
ip = &8;  /* oops -- you can't take the address of a literal */
```

(202) only lvalues may be assigned to or modified *(Parser)*

Only an lvalue (i.e. an identifier or expression directly denoting addressable storage) can be assigned to or otherwise modified, e.g.:

```
int array[10];
int * ip;
char c;
array = ip;   /* array isn't a variable,
                 it can't be written to */
```

A typecast does not yield an lvalue, e.g.:

```
/* the contents of c cast to int
   is only a intermediate value */
(int)c = 1;
```

However you can write this using pointers:

```
*(int *)&c = 1
```

(203) illegal operation on bit variable *(Parser)*

Not all operations on `bit` variables are supported. This operation is one of those, e.g.:

```
bit   b;
int * ip;
ip = &b;  /* oops --
             cannot take the address of a bit object */
```

(204) void function can't return a value *(Parser)*

A void function cannot return a value. Any `return` statement should not be followed by an expression, e.g.:

```
void run(void)
{
  step();
  return 1;
  /* either run should not be void, or remove the 1 */
}
```

(205) integral type required *(Parser)*

This operator requires operands that are of integral type only.

(206) illegal use of void expression *(Parser)*

A `void` expression has no value and therefore you can't use it anywhere an expression with a value is required, e.g. as an operand to an arithmetic operator.

(207) simple type required for "*" *(Parser)*

A simple type (i.e. not an array or structure) is required as an operand to this operator.

(208) operands of "*" not same type *(Parser)*

The operands of this operator are of different pointer, e.g.:

```
int * ip;
char * cp, * cp2;
cp = flag ? ip : cp2;
/* result of ? : will be int * or char * */
```

Maybe you meant something like:

```
cp = flag ? (char *)ip : cp2;
```

(209) type conflict *(Parser)*

The operands of this operator are of incompatible types.

(210) bad size list *(Parser)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(211) taking sizeof bit is illegal *(Parser)*

It is illegal to use the sizeof operator with the HI-TECH C bit type. When used against a type the sizeof operator gives the number of bytes required to store an object that type. Therefore its usage with the bit type make no sense and is an illegal operation.

(212) missing number after pragma "pack" *(Parser)*

The pragma pack requires a decimal number as argument. This specifies the alignment of each member within the structure. Use this with caution as some processors enforce alignment and will not operate correctly if word fetches are made on odd boundaries, e.g.:

```
#pragma pack  /* what is the alignment value */
```

Maybe you meant something like:

```
#pragma pack 2
```

(214) missing number after pragma "interrupt_level" *(Parser)*

The pragma interrupt_level requires an argument from 0 to 7.

(215) missing argument to pragma "switch" *(Parser)*

The pragma switch requires an argument of auto, direct or simple, e.g.:

```
#pragma switch  /* oops -- this requires a switch mode */
```

maybe you meant something like:

```
#pragma switch simple
```

(216) missing argument to pragma "psect" *(Parser)*

The pragma psect requires an argument of the form *oldname* = *newname* where *oldname* is an existing psect name known to the compiler, and *newname* is the desired new name, e.g.:

```
#pragma psect  /* oops -- this requires an psect to redirect */
```

maybe you meant something like:

```
#pragma psect text=specialtext
```

(218) missing name after pragma "inline" *(Parser)*

The `inline` pragma expects the name of a function to follow. The function name must be recognized by the code generator for it to be expanded; other functions are not altered, e.g.:

```
#pragma inline  /* what is the function name? */
```

maybe you meant something like:

```
#pragma inline memcpy
```

(219) missing name after pragma "printf_check" *(Parser)*

The `printf_check` pragma expects the name of a function to follow. This specifies printf-style format string checking for the function, e.g.

```
#pragma printf_check  /* what function is to be checked? */
```

Maybe you meant something like:

```
#pragma printf_check sprintf
```

Pragmas for all the standard printf-like function are already contained in `<stdio.h>`.

(220) exponent expected *(Parser)*

A floating point constant must have at least one digit after the `e` or `E`., e.g.:

```
float f;
f = 1.234e;  /* oops -- what is the exponent? */
```

(221) hexadecimal digit expected *(Parser)*

After `0x` should follow at least one of the HEX digits `0-9` and `A-F` or `a-f` , e.g.:

```
a = 0xg6;  /* oops -- was that meant to be a = 0xf6 ? */
```

(222) binary digit expected *(Parser)*

A binary digit was expected following the `0b` format specifier, e.g.

```
i = 0bf000;  /* oops -- f000 is not a base two value */
```

(223) digit out of range *(Parser, Assembler)*

A digit in this number is out of range of the radix for the number, e.g. using the digit 8 in an octal number, or HEX digits A-F in a decimal number. An octal number is denoted by the digit string commencing with a zero, while a HEX number starts with "0X" or "0x". For example:

```
int a = 058;
/* leading 0 implies octal which has digits 0 - 7 */
```

(224) illegal "#" directive *(Parser)*

An illegal `#` preprocessor has been detected. Likely a directive has been misspelled in your code somewhere.

(225) missing character in character constant *(Parser)*

The character inside the single quotes is missing, e.g.:

```
char c = '';  /* the character value of what? */
```

(226) char const too long *(Parser)*

A character constant enclosed in single quotes may not contain more than one character, e.g.:

```
c = '12';  /* oops -- only one character may be specified */
```

(227) "." expected after ".." *(Parser)*

The only context in which two successive dots may appear is as part of the *ellipsis* symbol, which must have 3 dots. (An *ellipsis* is used in function prototypes to indicate a variable number of parameters.)

Either `..` was meant to be an *ellipsis* symbol which would require you to add an extra *dot*, or it was meant to be a *structure member operator* which would require you remove one *dot*.

(228) illegal character (*) *(Parser)*

This character is illegal in the C code. Valid characters are the letters, digits and those comprising the acceptable operators, e.g.:

```
c = a;  /* oops -- did you mean c = 'a'; ? */
```

(229) unknown qualifier "*" given to -A *(Parser)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(230) missing argument to -A *(Parser)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(231) unknown qualifier "*" given to -I *(Parser)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(232) missing argument to -I *(Parser)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(233) bad -Q option "*" *(Parser)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(234) close error *(Parser)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(236) simple integer expression required *(Parser)*

A simple integral expression is required after the operator `@` , used to associate an absolute address with a variable, e.g.:

```
int address;
char LOCK @ address;
```

(237) function "*" redefined *(Parser)*

More than one definition for a function has been encountered in this module. Function overloading is illegal, e.g.:

```
int twice(int a)
{
  return a*2;
}
/* only one prototype & definition of rv can exist */
long twice(long a)
{
  return a*2;
}
```

(238) illegal initialisation *(Parser)*

You can't initialize a `typedef` declaration, because it does not reserve any storage that can be initialized, e.g.:

```
/* oops -- uint is a type, not a variable */
typedef unsigned int uint = 99;
```

(239) identifier "*" redefined (from line *) *(Parser)*

This identifier has already been defined in the same scope. It cannot be defined again, e.g.:

```
  int a;  /* a filescope variable called "a" */
  int a;  /* attempting to define another of the same name */
```

Note that variables with the same name, but defined with different scopes are legal, but not recommended.

(240) too many initializers *(Parser)*

There are too many initializers for this object. Check the number of initializers against the object definition (array or structure), e.g.:

```
/* three elements, but four initializers */
int ivals[3] = { 2, 4, 6, 8};
```

(241) initialization syntax *(Parser)*

The initialization of this object is syntactically incorrect. Check for the correct placement and number of braces and commas, e.g.:

```
int iarray[10] = {{'a', 'b', 'c'};
/* oops -- one two many {s */
```

(242) illegal type for switch expression *(Parser)*

A `switch` operation must have an expression that is either an integral type or an enumerated value, e.g:

```
double d;
switch(d) {  /* oops -- this must be integral */
  case '1.0':
    d = 0;
}
```

(243) inappropriate break/continue *(Parser)*

A `break` or `continue` statement has been found that is not enclosed in an appropriate control structure. A `continue` can only be used inside a `while`, `for` or `do while` loop, while `break` can only be used inside those loops or a `switch` statement, e.g.:

```
switch(input) {
  case 0:
    if(output == 0)
      input = 0xff;
    } /* oops! this shouldn't be here and closed the switch */
    break;      /* this should be inside the switch */
```

(244) "default" case redefined *(Parser)*

There is only allowed to be one `default` label in a switch statement. You have more than one, e.g.:

```
switch(a) {
default:      /* if this is the default case... */
  b = 9;
  break;
default:      /* then what is this? */
  b = 10;
  break;
```

(245) "default" case not in switch *(Parser)*

A label has been encountered called `default` but it is not enclosed by a `switch` statement. A `default` label is only legal inside the body of a `switch` statement.

If there is a `switch` statement before this `default` label, there may be one too many closing braces in the `switch` code which would prematurely terminate the `switch` statement. See message 246.

(246) case label not in switch *(Parser)*

A `case` label has been encountered, but there is no enclosing `switch` statement. A `case` label may only appear inside the body of a `switch` statement.

If there is a `switch` statement before this `case` label, there may be one too many closing braces in the `switch` code which would prematurely terminate the `switch` statement, e.g.:

```
switch(input) {
  case '0':
    count++;
    break;
  case '1':
    if(count>MAX)
      count= 0;
    }            /* oops -- this shouldn't be here */
    break;
  case '2':    /* error flagged here */
```

(247) duplicate label "*" *(Parser)*

The same name is used for a label more than once in this function. Note that the scope
of labels is the entire function, not just the block that encloses a label, e.g.:

```
start:
  if(a > 256)
    goto end;
start:            /* error flagged here */
  if(a == 0)
    goto start;  /* which start label do I jump to? */
```

(248) inappropriate "else" *(Parser)*

An `else` keyword has been encountered that cannot be associated with an `if`
statement. This may mean there is a missing brace or other syntactic error, e.g.:

```
/* here is a comment which I have forgotten to close...
if(a > b) {
  c = 0;
/* ... that will be closed here, thus removing the "if" */
else        /* my "if" has been lost */
  c = 0xff;
```

(249) probable missing "}" in previous block *(Parser)*

The compiler has encountered what looks like a function or other declaration, but the
preceding function has not been ended with a closing brace. This probably means that
a closing brace has been omitted from somewhere in the previous function, although it
may well not be the last one, e.g.:

```
void set(char a)
{
  PORTA = a;
                  /* the closing brace was left out here */
void clear(void)  /* error flagged here */
{
  PORTA = 0;
}
```

(251) array dimension redeclared *(Parser)*

An array dimension has been declared as a different non-zero value from its previous
declaration. It is acceptable to redeclare the size of an array that was previously
declared with a zero dimension, but not otherwise, e.g.:

```
extern int array[5];
int array[10];        /* oops -- has it 5 or 10 elements? */
```

(252) argument * conflicts with prototype *(Parser)*

The argument specified (argument 0 is the left most argument) of this function definition
does not agree with a previous prototype for this function, e.g.:

```
/* this is supposedly calc's prototype */
extern int calc(int, int);
int calc(int a, long int b)  /* hmmm -- which is right? */
{                            /* error flagged here */
  return sin(b/a);
}
```

(253) argument list conflicts with prototype *(Parser)*

The argument list in a function definition is not the same as a previous prototype for that function. Check that the number and types of the arguments are all the same.

```
extern int calc(int);   /* this is supposedly calc's prototype */
int calc(int a, int b)  /* hmmm -- which is right? */
{                       /* error flagged here */
  return a + b;
}
```

(254) undefined *: "*" *(Parser)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(255) not a member of the struct/union "*" *(Parser)*

This identifier is not a member of the structure or union type with which it used here, e.g.:

```
struct {
  int a, b, c;
} data;
if(data.d)    /* oops --
                there is no member d in this structure */
  return;
```

(256) too much indirection *(Parser)*

A pointer declaration may only have 16 levels of indirection.

(257) only "register" storage class allowed *(Parser)*

The only storage class allowed for a function parameter is `register`, e.g.:

```
void process(static int input)
```

(258) duplicate qualifier *(Parser)*

There are two occurrences of the same qualifier in this type specification. This can occur either directly or through the use of a typedef. Remove the redundant qualifier. For example:

```
typedef volatile int vint;
/* oops -- this results in two volatile qualifiers */
volatile vint very_vol;
```

(259) can't be qualified both far and near *(Parser)*

It is illegal to qualify a type as both `far` and `near`, e.g.:

```
far near int spooky;  /* oops -- choose far or near, not both */
```

(260) undefined enum tag "*" *(Parser)*

This enum tag has not been defined, e.g.:

```
enum WHAT what;  /* a definition for WHAT was never seen */
```

(261) struct/union member "*" redefined *(Parser)*

This name of this member of the struct or union has already been used in this `struct` or `union`, e.g.:

```
struct {
  int a;
  int b;
  int a;  /* oops -- a different name is required here */
} input;
```

(262) struct/union "*" redefined *(Parser)*

A structure or union has been defined more than once, e.g.:

```
struct {
  int a;
} ms;
struct {
  int a;
} ms;    /* was this meant to be the same name as above? */
```

(263) members can't be functions *(Parser)*

A member of a structure or a union may not be a function. It may be a pointer to a function, e.g.:

```
struct {
  int a;
  int get(int);  /* should be a pointer: int (*get)(int); */
} object;
```

(264) bad bitfield type *(Parser)*

A bitfield may only have a type of `int` (`signed` or `unsigned`), e.g.:

```
struct FREG {
  char b0:1;   /* these must be part of an int, not char */
  char   :6;
  char b7:1;
} freg;
```

(265) integer constant expected *(Parser)*

A *colon* appearing after a member name in a structure declaration indicates that the member is a bitfield. An integral constant must appear after the *colon* to define the number of bits in the bitfield, e.g.:

```
struct {
  unsigned first:  /* oops -- should be: unsigned first; */
  unsigned second;
} my_struct;
```

If this was meant to be a structure with bitfields, then the following illustrates an example:

```
struct {
  unsigned first : 4;  /* 4 bits wide */
  unsigned second: 4;  /* another 4 bits */
} my_struct;
```

(266) storage class illegal   *(Parser)*

A structure or union member may not be given a storage class. Its storage class is determined by the storage class of the structure, e.g.:

```
struct {
  /* no additional qualifiers may be present with members */
  static int first;
} ;
```

(267) bad storage class   *(Code Generator)*

The code generator has encountered a variable definition whose storage class is invalid, e.g.:

```
auto int foo; /* auto not permitted with global variables */
int power(static int a)  /* parameters may not be static */
{
  return foo * a;
}
```

(268) inconsistent storage class   *(Parser)*

A declaration has conflicting storage classes. Only one storage class should appear in a declaration, e.g.:

```
extern static int where;  /* so is it static or extern? */
```

(269) inconsistent type   *(Parser)*

Only one basic type may appear in a declaration, e.g.:

```
int float if;  /* is it int or float? */
```

(270) variable can't have storage class "register"   *(Parser)*

Only function parameters or `auto` variables may be declared using the `register` qualifier, e.g.:

```
register int gi;      /* this cannot be qualified register */
int process(register int input)  /* this is okay */
{
  return input + gi;
}
```

(271) type can't be long   *(Parser)*

Only `int` and `float` can be qualified with `long`.

```
long char lc;  /* what? */
```

(272) type can't be short   *(Parser)*

Only `int` can be modified with `short`, e.g.:

```
short float sf;  /* what? */
```

(273) type can't be both signed and unsigned   *(Parser)*

The type modifiers `signed` and `unsigned` cannot be used together in the same declaration, as they have opposite meaning, e.g.:

```
signed unsigned int confused;  /* which is it? */
```

(274) type can't be unsigned   *(Parser)*

A floating point type cannot be made `unsigned`, e.g.:

```
unsigned float uf;  /* what? */
```

(275) "..." illegal in non-prototype argument list *(Parser)*

The *ellipsis* symbol may only appear as the last item in a prototyped argument list. It may not appear on its own, nor may it appear after argument names that do not have types, i.e. K&R-style non-prototype function definitions. For example:

```
/* K&R-style non-prototyped function definition */
int kandr(a, b, ...)
  int a, b;
{
```

(276) type specifier required for prototyped argument *(Parser)*

A type specifier is required for a prototyped argument. It is not acceptable to just have an identifier.

(277) can't mix prototyped and non-prototyped arguments *(Parser)*

A function declaration can only have all prototyped arguments (i.e. with types inside the parentheses) or all K&R style args (i.e. only names inside the parentheses and the argument types in a declaration list before the start of the function body), e.g.:

```
int plus(int a, b)  /* oops -- a is prototyped, b is not */
int b;
{
  return a + b;
}
```

(278) argument "*" redeclared *(Parser)*

The specified argument is declared more than once in the same argument list, e.g.

```
/* can't have two parameters called "a" */
int calc(int a, int a)
```

(279) initialization of function arguments is illegal *(Parser)*

A function argument can't have an initializer in a declaration. The initialization of the argument happens when the function is called and a value is provided for the argument by the calling function, e.g.:

```
/* oops -- a is initialized when proc is called */
extern int proc(int a = 9);
```

(280) arrays of functions are illegal *(Parser)*

You can't define an array of functions. You can however define an array of pointers to functions, e.g.:

```
int * farray[]();  /* oops -- should be: int (* farray[])(); */
```

(281) functions can't return functions *(Parser)*

A function cannot return a function. It can return a function pointer. A function returning a pointer to a function could be declared like this: int (* (name()))(). Note the many parentheses that are necessary to make the parts of the declaration bind correctly.

(282) functions can't return arrays *(Parser)*

A function can return only a scalar (simple) type or a structure. It cannot return an array.

(283) dimension required *(Parser)*

Only the most significant (i.e. the first) dimension in a multi-dimension array may not be assigned a value. All succeeding dimensions must be present as a constant expression, e.g.:

```
/* This should be, e.g.: int arr[][7] */
int get_element(int arr[2][])
{
  return array[1][6];
}
```

(284) invalid dimension *(Parser)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(285) no identifier in declaration *(Parser)*

The identifier is missing in this declaration. This error can also occur where the compiler has been confused by such things as missing closing braces, e.g.:

```
void interrupt(void)  /* what is the name of this function? */
{
}
```

(286) declarator too complex *(Parser)*

This declarator is too complex for the compiler to handle. Examine the declaration and find a way to simplify it. If the compiler finds it too complex, so will anybody maintaining the code.

(287) arrays of bits or pointers to bit are illegal *(Parser)*

It is not legal to have an array of bits, or a pointer to bit variable, e.g.:

```
bit barray[10];  /* wrong -- no bit arrays */
bit * bp;        /* wrong -- no pointers to bit variables */
```

(288) only functions may be void *(Parser)*

A variable may not be `void`. Only a function can be `void`, e.g.:

```
int a;
void b;  /* this makes no sense */
```

(289) only functions may be qualified "interrupt" *(Parser)*

The qualifier `interrupt` may not be applied to anything except a function, e.g.:

```
/* variables cannot be qualified interrupt */
interrupt int input;
```

(290) illegal function qualifier(s) *(Parser)*

A qualifier has been applied to a function which makes no sense in this context. Some qualifier only make sense when used with an lvalue, e.g. const or volatile. This may indicate that you have forgotten out a star * indicating that the function should return a pointer to a qualified object, e.g.

```
const char ccrv(void) /* const * char ccrv(void) perhaps? */
{                     /* error flagged here */
  return ccip;
}
```

(291) K&R identifier "*" not an argument *(Parser)*

This identifier that has appeared in a K&R style argument declarator is not listed inside the parentheses after the function name, e.g.:

```
int process(input)
int unput;          /* oops -- that should be int input; */
{
}
```

(292) function parameter may not be a function *(Parser)*

A function parameter may not be a function. It may be a pointer to a function, so perhaps a "*" has been omitted from the declaration.

(293) bad size in index_type() *(Parser)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(294) can't allocate * bytes of memory *(Code Generator, Hexmate)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(295) expression too complex *(Parser)*

This expression has caused overflow of the compiler's internal stack and should be re-arranged or split into two expressions.

(296) out of memory *(Objtohex)*

This could be an internal compiler error. Contact HI-TECH Software technical support with details.

(297) bad argument (*) to tysize() *(Parser)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(298) end of file in #asm *(Preprocessor)*

An end of file has been encountered inside a #asm block. This probably means the #endasm is missing or misspelled, e.g.:

```
#asm
  MOV  r0, #55
  MOV  [r1], r0
}               /* oops -- where is the #endasm */
```

(300) unexpected end of file *(Parser)*

An end-of-file in a C module was encountered unexpectedly, e.g.:

```
void main(void)
{
  init();
  run();    /* is that it? What about the close brace */
```

(301) end of file on string file *(Parser)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(302) can't reopen "*": * *(Parser)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(303) can't allocate * bytes of memory (line *) *(Parser)*

The parser was unable to allocate memory for the longest string encountered, as it attempts to sort and merge strings. Try reducing the number or length of strings in this module.

(306) can't allocate * bytes of memory for * *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(307) too many qualifier names *(Parser)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(308) too many case labels in switch *(Code Generator)*

There are too many `case` labels in this `switch` statement. The maximum allowable number of `case` labels in any one `switch` statement is 511.

(309) too many symbols *(Assembler)*

There are too many symbols for the assembler's symbol table. Reduce the number of symbols in your program.

(310) "]" expected *(Parser)*

A closing square bracket was expected in an array declaration or an expression using an array index, e.g.

```
process(carray[idx);  /* oops --
                    should be: process(carray[idx]); */
```

(311) closing quote expected *(Parser)*

A closing quote was expected for the indicated string.

(312) "*" expected *(Parser)*

The indicated token was expected by the parser.

(313) function body expected *(Parser)*

Where a function declaration is encountered with K&R style arguments (i.e. argument names but no types inside the parentheses) a function body is expected to follow, e.g.:

```
/* the function block must follow, not a semicolon */
int get_value(a, b);
```

(314) ";" expected *(Parser)*

A *semicolon* is missing from a statement. A close brace or keyword was found following a statement with no terminating *semicolon* , e.g.:

```
while(a) {
  b = a--  /* oops -- where is the semicolon? */
}          /* error is flagged here */
```

Note: Omitting a semicolon from statements not preceding a close brace or keyword typically results in some other error being issued for the following code which the parser assumes to be part of the original statement.

(315) "{" expected *(Parser)*

An *opening brace* was expected here. This error may be the result of a function definition missing the *opening brace* , e.g.:

```
/* oops! no opening brace after the prototype */
void process(char c)
  return max(c, 10) * 2; /* error flagged here */
}
```

(316) "}" expected *(Parser)*

A *closing brace* was expected here. This error may be the result of a initialized array missing the *closing brace* , e.g.:

```
char carray[4] = { 1, 2, 3, 4;  /* oops -- no closing brace */
```

(317) "(" expected *(Parser)*

An *opening parenthesis* , (, was expected here. This must be the first token after a while , for , if , do or asm keyword, e.g.:

```
if a == b  /* should be: if(a == b) */
  b = 0;
```

(318) string expected *(Parser)*

The operand to an asm statement must be a string enclosed in parentheses, e.g.:

```
asm(nop);  /* that should be asm("nop"); */
```

(319) while expected *(Parser)*

The keyword while is expected at the end of a do statement, e.g.:

```
do {
  func(i++);
}             /* do the block while what condition is true? */
if(i > 5)     /* error flagged here */
  end();
```

(320) ":" expected *(Parser)*

A *colon* is missing after a case label, or after the keyword default. This often occurs when a *semicolon* is accidentally typed instead of a *colon* , e.g.:

```
switch(input) {
  case 0;          /* oops -- that should have been: case 0: */
    state = NEW;
```

(321) label identifier expected *(Parser)*

An identifier denoting a label must appear after `goto` , e.g.:

```
if(a)
  goto 20;
/* this is not BASIC -- a valid C label must follow a goto */
```

(322) enum tag or "{" expected *(Parser)*

After the keyword `enum` must come either an identifier that is or will be defined as an `enum` tag, or an opening brace, e.g.:

```
enum 1, 2;  /* should be, e.g.: enum {one=1, two }; */
```

(323) struct/union tag or "{" expected *(Parser)*

An identifier denoting a structure or union or an opening brace must follow a `struct` or `union` keyword, e.g.:

```
struct int a;  /* this is not how you define a structure */
```

You might mean something like:

```
struct {
  int a;
} my_struct;
```

(324) too many arguments for printf-style format string *(Parser)*

There are too many arguments for this format string. This is harmless, but may represent an incorrect format string, e.g.:

```
/* oops -- missed a placeholder? */
printf("%d - %d", low, high, median);
```

(325) error in printf-style format string *(Parser)*

There is an error in the format string here. The string has been interpreted as a `printf()` style format string, and it is not syntactically correct. If not corrected, this will cause unexpected behavior at run time, e.g.:

```
printf("%l", lll);  /* oops -- maybe: printf("%ld", lll); */
```

(326) long int argument required in printf-style format string *(Parser)*

A long argument is required for this format specifier. Check the number and order of format specifiers and corresponding arguments, e.g.:

```
printf("%lx", 2);  // maybe you meant: printf("%lx", 2L);
```

(327) long long int argument required in printf-style format string *(Parser)*

A long long argument is required for this format specifier. Check the number and order of format specifiers and corresponding arguments, e.g.:

```
printf("%llx", 2);  // maybe you meant: printf("%llx", 2LL);
```

Note that not all HI-TECH C compilers provide support for a long long integer type.

(328) int argument required in printf-style format string *(Parser)*

An integral argument is required for this printf-style format specifier. Check the number and order of format specifiers and corresponding arguments, e.g.:

```
printf("%d", 1.23); /* wrong number or wrong placeholder */
```

(329) double argument required in printf-style format string *(Parser)*

The printf format specifier corresponding to this argument is `%f` or similar, and requires a floating point expression. Check for missing or extra format specifiers or arguments to printf.

```
printf("%f", 44);  /* should be: printf("%f", 44.0); */
```

(330) pointer to * argument required in printf-style format string *(Parser)*

A pointer argument is required for this format specifier. Check the number and order of format specifiers and corresponding arguments.

(331) too few arguments for printf-style format string *(Parser)*

There are too few arguments for this format string. This would result in a garbage value being printed or converted at run time, e.g.:

```
printf("%d - %d", low);
  /* oops! where is the other value to print? */
```

(332) "interrupt_level" should be 0 to 7 *(Parser)*

The pragma `interrupt_level` must have an argument from 0 to 7, e.g.:

```
#pragma interrupt_level  /* oops -- what is the level */
void interrupt isr(void)
{
  /* isr code goes here */
}
```

(333) unrecognized qualifier name after "strings" *(Parser)*

The `pragma strings` was passed a qualifier that was not identified, e.g.:

```
/* oops -- should that be #pragma strings const ? */
#pragma strings cinst
```

(334) unrecognized qualifier name after "printf_check" *(Parser)*

The `#pragma printf_check` was passed a qualifier that could not be identified, e.g.:

```
/* oops -- should that be const not cinst? */
#pragma printf_check(printf) cinst
```

(335) unknown pragma "*" *(Parser)*

An unknown `pragma` directive was encountered, e.g.:

```
#pragma rugsused w  /* I think you meant regsused */
```

(336) string concatenation across lines *(Parser)*

Strings on two lines will be concatenated. Check that this is the desired result, e.g.:

```
char * cp = "hi"
  "there";    /* this is okay,
                  but is it what you had intended? */
```

(337) line does not have a newline on the end *(Parser)*

The last line in the file is missing the *newline* (operating system dependent character) from the end. Some editors will create such files, which can cause problems for include files. The ANSI C standard requires all source files to consist of complete lines only.

(338) can't create * file "*" *(Any)*

The application tried to create or open the named file, but it could not be created. Check that all file pathnames are correct.

(339) initializer in extern declaration *(Parser)*

A declaration containing the keyword `extern` has an initializer. This overrides the `extern` storage class, since to initialise an object it is necessary to define (i.e. allocate storage for) it, e.g.:

```
extern int other = 99;  /* if it's extern and not allocated
                           storage, how can it be initialized? */
```

(340) string not terminated by null character. *(Parser)*

A char array is being initialized with a string literal larger than the array. Hence there is insufficient space in the array to safely append a null terminating character, e.g.:

```
char foo[5] = "12345"; /* the string stored in foo won't have
                           a null terminating, i.e.
                           foo = ['1', '2', '3', '4', '5'] */
```

(343) implicit return at end of non-void function *(Parser)*

A function which has been declared to return a value has an execution path that will allow it to reach the end of the function body, thus returning without a value. Either insert a `return` statement with a value, or if the function is not to return a value, declare it `void`, e.g.:

```
int mydiv(double a, int b)
{
  if(b != 0)
    return a/b;    /* what about when b is 0? */
}                  /* warning flagged here */
```

(344) non-void function returns no value *(Parser)*

A function that is declared as returning a value has a `return` statement that does not specify a return value, e.g.:

```
int get_value(void)
{
  if(flag)
    return val++;
  return;
  /* what is the return value in this instance? */
}
```

(345) unreachable code *(Parser)*

This section of code will never be executed, because there is no execution path by which it could be reached, e.g.:

```
while(1)            /* how does this loop finish? */
  process();
flag = FINISHED;  /* how do we get here? */
```

(346) declaration of "*" hides outer declaration *(Parser)*

An object has been declared that has the same name as an outer declaration (i.e. one outside and preceding the current function or block). This is legal, but can lead to accidental use of one variable when the outer one was intended, e.g.:

```
int input;          /* input has filescope */
void process(int a)
{
  int input;        /* local blockscope input */
  a = input;        /* this will use the local variable.
                        Is this right? */
```

(347) external declaration inside function *(Parser)*

A function contains an `extern` declaration. This is legal but is invariably not desirable as it restricts the scope of the function declaration to the function body. This means that if the compiler encounters another declaration, use or definition of the extern object later in the same file, it will no longer have the earlier declaration and thus will be unable to check that the declarations are consistent. This can lead to strange behavior of your program or signature errors at link time. It will also hide any previous declarations of the same thing, again subverting the compiler's type checking. As a general rule, always declare `extern` variables and functions outside any other functions. For example:

```
int process(int a)
{
  /* this would be better outside the function */
  extern int away;
  return away + a;
}
```

(348) auto variable "*" should not be qualified *(Parser)*

An `auto` variable should not have qualifiers such as `near` or `far` associated with it. Its storage class is implicitly defined by the stack organization. An `auto` variable may be qualified with `static` , but it is then no longer `auto`.

(349) non-prototyped function declaration for "*" *(Parser)*

A function has been declared using old-style (K&R) arguments. It is preferable to use prototype declarations for all functions, e.g.:

```
int process(input)
int input;          /* warning flagged here */
{
}
```

This would be better written:

```
int process(int input)
{
}
```

(350) unused * "*" (from line *) *(Parser)*

The indicated object was never used in the function or module being compiled. Either this object is redundant, or the code that was meant to use it was excluded from compilation or misspelled the name of the object. Note that the symbols `rcsid` and `sccsid` are never reported as being unused.

(352) float parameter coerced to double *(Parser)*

Where a non-prototyped function has a parameter declared as `float`, the compiler converts this into a `double float`. This is because the default C type conversion conventions provide that when a floating point number is passed to a non-prototyped function, it will be converted to `double`. It is important that the function declaration be consistent with this convention, e.g.:

```
double inc_flt(f)  /* f will be converted to double */
float f;           /* warning flagged here */
{
  return f * 2;
}
```

(353) sizeof external array "*" is zero *(Parser)*

The size of an external array evaluates to zero. This is probably due to the array not having an explicit dimension in the extern declaration.

(354) possible pointer truncation *(Parser)*

A pointer qualified far has been assigned to a default pointer or a pointer qualified near, or a default pointer has been assigned to a pointer qualified near. This may result in truncation of the pointer and loss of information, depending on the memory model in use.

(355) implicit signed to unsigned conversion *(Parser)*

A `signed` number is being assigned or otherwise converted to a larger `unsigned` type. Under the ANSI C "value preserving" rules, this will result in the `signed` value being first sign-extended to a `signed` number the size of the target type, then converted to `unsigned` (which involves no change in bit pattern). Thus an unexpected sign extension can occur. To ensure this does not happen, first convert the signed value to an unsigned equivalent, e.g.:

```
signed char sc;
unsigned int ui;
ui = sc;    /* if sc contains 0xff,
              ui will contain 0xffff for example */
```

will perform a sign extension of the `char` variable to the longer type. If you do not want this to take place, use a cast, e.g.:

```
ui = (unsigned char)sc;
```

(356) implicit conversion of float to integer *(Parser)*

A floating point value has been assigned or otherwise converted to an integral type. This could result in truncation of the floating point value. A typecast will make this warning go away.

```
double dd;
int i;
i = dd;    /* is this really what you meant? */
```

If you do intend to use an expression like this, then indicate that this is so by a cast:

```
i = (int)dd;
```

(357) illegal conversion of integer to pointer *(Parser)*

An integer has been assigned to or otherwise converted to a pointer type. This will usually mean you have used the wrong variable, but if this is genuinely what you want to do, use a typecast to inform the compiler that you want the conversion and the warning will be suppressed. This may also mean you have forgotten the `&` address operator, e.g.:

```
int * ip;
int i;
ip = i;     /* oops -- did you mean ip = &i ? */
```

If you do intend to use an expression like this, then indicate that this is so by a cast:

```
ip = (int *)i;
```

(358) illegal conversion of pointer to integer *(Parser)*

A pointer has been assigned to or otherwise converted to a integral type. This will usually mean you have used the wrong variable, but if this is genuinely what you want to do, use a typecast to inform the compiler that you want the conversion and the warning will be suppressed. This may also mean you have forgotten the `*` dereference operator, e.g.:

```
int * ip;
int i;
i = ip;     /* oops -- did you mean i = *ip ? */
```

If you do intend to use an expression like this, then indicate that this is so by a cast:

```
i = (int)ip;
```

(359) illegal conversion between pointer types *(Parser)*

A pointer of one type (i.e. pointing to a particular kind of object) has been converted into a pointer of a different type. This will usually mean you have used the wrong variable, but if this is genuinely what you want to do, use a typecast to inform the compiler that you want the conversion and the warning will be suppressed, e.g.:

```
long input;
char * cp;
cp = &input;  /* is this correct? */
```

This is common way of accessing bytes within a multi-byte variable. To indicate that this is the intended operation of the program, use a cast:

```
cp = (char *)&input;  /* that's better */
```

This warning may also occur when converting between pointers to objects which have the same type, but which have different qualifiers, e.g.:

```
char * cp;
/* yes, but what sort of characters? */
cp = "I am a string of characters";
```

If the default type for string literals is `const char *`, then this warning is quite valid. This should be written:

```
const char * cp;
cp = "I am a string of characters";  /* that's better */
```

Omitting a qualifier from a pointer type is often disastrous, but almost certainly not what you intend.

(360) array index out of bounds *(Parser)*

An array is being indexed with a constant value that is less than zero, or greater than or equal to the number of elements in the array. This warning will not be issued when accessing an array element via a pointer variable, e.g.:

```
int i, * ip, input[10];
i = input[-2];            /* oops -- this element doesn't exist */
ip = &input[5];
i = ip[-2];               /* this is okay */
```

(361) function declared implicit int *(Parser)*

Where the compiler encounters a function call of a function whose name is presently undefined, the compiler will automatically declare the function to be of type int , with unspecified (K&R style) parameters. If a definition of the function is subsequently encountered, it is possible that its type and arguments will be different from the earlier implicit declaration, causing a compiler error. The solution is to ensure that all functions are defined or at least declared before use, preferably with prototyped parameters. If it is necessary to make a forward declaration of a function, it should be preceded with the keywords extern or static as appropriate. For example:

```
/* I may prevent an error arising from calls below */
void set(long a, int b);
void main(void)
{
  /* by here a prototype for set should have seen */
  set(10L, 6);
}
```

(362) redundant "&" applied to array *(Parser)*

The address operator & has been applied to an array. Since using the name of an array gives its address anyway, this is unnecessary and has been ignored, e.g.:

```
int array[5];
int * ip;
/* array is a constant, not a variable; the & is redundant. */
ip = &array;
```

(363) redundant "&" or "*" applied to function address *(Parser)*

The address operator "&" has been applied to a function. Since using the name of a function gives its address anyway, this is unnecessary and has been ignored, e.g.:

```
extern void foo(void);
void main(void)
{
    void(*bar)(void);
    /* both assignments are equivalent */
    bar = &foo;
    bar = foo;  /* the & is redundant */
}
```

(364) attempt to modify object qualified * *(Parser)*

Objects declared const or code may not be assigned to or modified in any other way by your program. The effect of attempting to modify such an object is compiler-specific.

```
const int out = 1234;  /* "out" is read only */
out = 0;               /* oops --
                           writing to a read-only object */
```

(365) pointer to non-static object returned *(Parser)*

This function returns a pointer to a non-`static` (e.g. `auto`) variable. This is likely to be an error, since the storage associated with automatic variables becomes invalid when the function returns, e.g.:

```
char * get_addr(void)
{
  char c;
  /* returning this is dangerous;
     the pointer could be dereferenced */
  return &c;
}
```

(366) operands of "*" not same pointer type *(Parser)*

The operands of this operator are of different pointer types. This probably means you have used the wrong pointer, but if the code is actually what you intended, use a typecast to suppress the error message.

(367) identifier is already extern; can't be static *(Parser)*

This function was already declared `extern`, possibly through an implicit declaration. It has now been redeclared `static`, but this redeclaration is invalid.

```
void main(void)
{
  /* at this point the compiler assumes set is extern... */
  set(10L, 6);
}
/* now it finds out otherwise */
static void set(long a, int b)
{
  PORTA = a + b;
}
```

(368) array dimension on "*[]" ignored *(Preprocessor)*

An array dimension on a function parameter has been ignored because the argument is actually converted to a pointer when passed. Thus arrays of any size may be passed. Either remove the dimension from the parameter, or define the parameter using pointer syntax, e.g.:

```
/* param should be: "int array[]" or "int *" */
int get_first(int array[10])
{                                   /* warning flagged here */
  return array[0];
}
```

(369) signed bitfields not supported *(Parser)*

Only `unsigned` bitfields are supported. If a bitfield is declared to be type `int`, the compiler still treats it as `unsigned`, e.g.:

```
struct {
  signed int sign: 1;    /* this must be unsigned */
  signed int value: 15;
} ;
```

(370) illegal basic type; int assumed *(Parser)*

The basic type of a cast to a qualified basic type couldn't not be recognized and the basic type was assumed to be `int`, e.g.:

```
/* here ling is assumed to be int */
unsigned char bar = (unsigned ling) 'a';
```

(371) missing basic type; int assumed *(Parser)*

This declaration does not include a basic type, so `int` has been assumed. This declaration is not illegal, but it is preferable to include a basic type to make it clear what is intended, e.g.:

```
char c;
i;      /* don't let the compiler make assumptions, use : int i */
func(); /* ditto, use: extern int func(int); */
```

(372) "," expected *(Parser)*

A *comma* was expected here. This could mean you have left out the *comma* between two identifiers in a declaration list. It may also mean that the immediately preceding type name is misspelled, and has thus been interpreted as an identifier, e.g.:

```
unsigned char a;
/* thinks: chat & b are unsigned, but where is the comma? */
unsigned chat b;
```

(373) implicit signed to unsigned conversion *(Parser)*

An `unsigned` type was expected where a `signed` type was given and was implicitly cast to `unsigned`, e.g.:

```
unsigned int foo = -1;
/* the above initialization is implicitly treated as:
   unsigned int foo = (unsigned) -1; */
```

(374) missing basic type; int assumed *(Parser)*

The basic type of a cast to a qualified basic type was missing and assumed to be `int.`, e.g.:

```
int i = (signed) 2; /* (signed) assumed to be (signed int) */
```

(375) unknown FNREC type "*" *(Linker)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(376) bad non-zero node in call graph *(Linker)*

The linker has encountered a top level node in the call graph that is referenced from lower down in the call graph. This probably means the program has indirect recursion, which is not allowed when using a compiled stack.

(378) can't create * file "*" *(Hexmate)*

This type of file could not be created. Is the file or a file by this name already in use?

(379) bad record type "*" *(Linker)*

This is an internal compiler error. Ensure the object file is a valid HI-TECH object file. Contact HI-TECH Software technical support with details.

(380) unknown record type (*)   *(Linker)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(381) record "*" too long (*)   *(Linker)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(382) incomplete record: type = *, length = * *(Dump, Xstrip)*

This message is produced by the DUMP or XSTRIP utilities and indicates that the object file is not a valid HI-TECH object file, or that it has been truncated. Contact HI-TECH Support with details.

(383) text record has length (*) too small   *(Linker)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(384) assertion failed: file *, line *, expression *   *(Linker, Parser)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(387) illegal or too many -G options   *(Linker)*

There has been more than one linker -g option, or the -g option did not have any arguments following. The arguments specify how the segment addresses are calculated.

(388) duplicate -M option   *(Linker)*

The map file name has been specified to the linker for a second time. This should not occur if you are using a compiler driver. If invoking the linker manually, ensure that only one instance of this option is present on the command line. See **Section 2.7.8 "-M: Generate Map File"** for information on the correct syntax for this option.

(389) illegal or too many -O options   *(Linker)*

This linker -o flag is illegal, or another -o option has been encountered. A -o option to the linker must be immediately followed by a filename with no intervening space.

(390) missing argument to -P   *(Linker)*

There have been too many -p options passed to the linker, or a -p option was not followed by any arguments. The arguments of separate -p options may be combined and separated by *commas*.

(391) missing argument to -Q   *(Linker)*

The -Q linker option requires the machine type for an argument.

(392) missing argument to -U   *(Linker)*

The -U (undefine) option needs an argument.

(393) missing argument to -W   *(Linker)*

The -W option (listing width) needs a numeric argument.

(394) duplicate -D or -H option *(Linker)*

The symbol file name has been specified to the linker for a second time. This should not occur if you are using a compiler driver. If invoking the linker manually, ensure that only one instance of either of these options is present on the command line.

(395) missing argument to -J *(Linker)*

The maximum number of errors before aborting must be specified following the $-j$ linker option.

(397) usage: hlink [-options] files.obj files.lib *(Linker)*

Improper usage of the command-line linker. If you are invoking the linker directly then please refer to Section **Section 5.2 "Operation"** for more details. Otherwise this may be an internal compiler error and you should contact HI-TECH Software technical support with details.

(398) output file can't be also an input file *(Linker)*

The linker has detected an attempt to write its output file over one of its input files. This cannot be done, because it needs to simultaneously read and write input and output files.

(400) bad object code format *(Linker)*

This is an internal compiler error. The object code format of an object file is invalid. Ensure it is a valid HI-TECH object file. Contact HI-TECH Software technical support with details.

(402) bad argument to -F *(Objtohex)*

The $-F$ option for objtohex has been supplied an invalid argument. If you are invoking this command-line tool directly then please refer to **Section 6.3 "Objtohex"** for more details. Otherwise this may be an internal compiler error and you should contact HI-TECH Software technical support with details.

(403) bad -E option: "*" *(Objtohex)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(404) bad maximum length value to -<digits> *(Objtohex)*

The first value to the OBJTOHEX $-n,m$ HEX length/rounding option is invalid.

(405) bad record size rounding value to -<digits> *(Objtohex)*

The second value to the OBJTOHEX $-n,m$ HEX length/rounding option is invalid.

(406) bad argument to -A *(Objtohex)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(407) bad argument to -U *(Objtohex)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(408) bad argument to -B *(Objtohex)*

This option requires an integer argument in either base 8, 10 or 16. If you are invoking `objtohex` directly then see **Section 6.3 "Objtohex"** for more details. Otherwise this may be an internal compiler error and you should contact HI-TECH Software technical support with details.

(409) bad argument to -P *(Objtohex)*

This option requires an integer argument in either base 8, 10 or 16. If you are invoking `objtohex` directly then see **Section 6.3 "Objtohex"** for more details. Otherwise this may be an internal compiler error and you should contact HI-TECH Software technical support with details.

(410) bad combination of options *(Objtohex)*

The combination of options supplied to `OBJTOHEX` is invalid.

(412) text does not start at 0 *(Objtohex)*

Code in some things must start at zero. Here it doesn't.

(413) write error on "*" *(Assembler, Linker, Cromwell)*

A write error occurred on the named file. This probably means you have run out of disk space.

(414) read error on "*" *(Linker)*

The linker encountered an error trying to read this file.

(415) text offset too low in COFF file *(Objtohex)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(416) bad character (*) in extended TEKHEX line *(Objtohex)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(417) seek error in "*" *(Linker)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(418) image too big *(Objtohex)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(419) object file is not absolute *(Objtohex)*

The object file passed to `OBJTOHEX` has relocation items in it. This may indicate it is the wrong object file, or that the linker or OBJTOHEX have been given invalid options. The object output files from the assembler are relocatable, not absolute. The object file output of the linker is absolute.

(420) too many relocation items *(Objtohex)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

---

(421) too many segments   *(Objtohex)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(422) no end record   *(Linker)*

This object file has no end record. This probably means it is not an object file. Contact HI-TECH Support if the object file was generated by the compiler.

(423) illegal record type   *(Linker)*

There is an error in an object file. This is either an invalid object file, or an internal error in the linker. Contact HI-TECH Support with details if the object file was created by the compiler.

(424) record too long   *(Objtohex)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(425) incomplete record   *(Objtohex, Libr)*

The object file passed to OBJTOHEX or the librarian is corrupted. Contact HI-TECH Support with details.

(427) syntax error in checksum list   *(Objtohex)*

There is a syntax error in a checksum list read by OBJTOHEX. The checksum list is read from standard input in response to an option.

(428) too many segment fixups   *(Objtohex)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(429) bad segment fixups   *(Objtohex)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(430) bad checksum specification   *(Objtohex)*

A checksum list supplied to OBJTOHEX is syntactically incorrect.

(431) bad argument to -E   *(Objtoexe)*

This option requires an integer argument in either base 8, 10 or 16. If you are invoking objtoexe directly then check this argument. Otherwise this may be an internal compiler error and you should contact HI-TECH Software technical support with details.

(432) usage: objtohex [-ssymfile] [object-file [exe-file]]   *(Objtohex)*

Improper usage of the command-line tool objtohex. If you are invoking objtohex directly then please refer to **Section 6.3 "Objtohex"** for more details. Otherwise this may be an internal compiler error and you should contact HI-TECH Software technical support with details.

(434) too many symbols (*)   *(Linker)*

There are too many symbols in the symbol table, which has a limit of * symbols. Change some global symbols to local symbols to reduce the number of symbols.

(435) bad segment selector "*" *(Linker)*

The segment specification option (-G) to the linker is invalid, e.g.:

`-GA/f0+10`

Did you forget the radix?

`-GA/f0h+10`

(436) psect "*" re-orged *(Linker)*

This psect has had its start address specified more than once.

(437) missing "=" in class spec *(Linker)*

A class spec needs an = sign, e.g. -Ctext=ROM See **Section "-Cpsect=class"** for more information.

(438) bad size in -S option *(Linker)*

The address given in a -S specification is invalid: it should be a valid number, in decimal, octal or hexadecimal radix. The radix is specified by a trailing O , for octal, or H for HEX. A leading 0x may also be used for hexadecimal. Case in not important for any number or radix. Decimal is the default, e.g.:

`-SCODE=f000`

Did you forget the radix?

`-SCODE=f000h`

(439) bad -D spec: "*" *(Linker)*

The format of a -D specification, giving a *delta* value to a class, is invalid, e.g.:

`-DCODE`

What is the *delta* value for this class? Maybe you meant something like:

`-DCODE=2`

(440) bad delta value in -D spec *(Linker)*

The *delta* value supplied to a -D specification is invalid. This value should an integer of base 8, 10 or 16.

(441) bad -A spec: "*" *(Linker)*

The format of a -A specification, giving address ranges to the linker, is invalid, e.g.:

`-ACODE`

What is the range for this class? Maybe you meant:

`-ACODE=0h-1fffh`

(442) missing address in -A spec *(Linker)*

The format of a -A specification, giving address ranges to the linker, is invalid, e.g.:

`-ACODE=`

What is the range for this class? Maybe you meant:

`-ACODE=0h-1fffh`

(443) bad low address "*" in -A spec *(Linker)*

The low address given in a -A specification is invalid: it should be a valid number, in decimal, octal or hexadecimal radix. The radix is specified by a trailing O (for octal) or H for HEX. A leading `0x` may also be used for hexadecimal. Case in not important for any number or radix. Decimal is default, e.g.:

```
-ACODE=1fff-3fffh
```

Did you forget the radix?

```
-ACODE=1fffh-3fffh
```

(444) expected "-" in -A spec *(Linker)*

There should be a minus sign, `-` , between the high and low addresses in a `-A` linker option, e.g.

```
-AROM=1000h
```

maybe you meant:

```
-AROM=1000h-1fffh
```

(445) bad high address "*" in -A spec *(Linker)*

The high address given in a `-A` specification is invalid: it should be a valid number, in decimal, octal or hexadecimal radix. The radix is specified by a trailing `O` , for octal, or `H` for HEX. A leading `0x` may also be used for hexadecimal. Case in not important for any number or radix. Decimal is the default, e.g.:

```
-ACODE=0h-ffff
```

Did you forget the radix?

```
-ACODE=0h-ffffh
```

See **Section 5.2.1 "-Aclass =low-high,..."** for more information.

(446) bad overrun address "*" in -A spec *(Linker)*

The overrun address given in a -A specification is invalid: it should be a valid number, in decimal, octal or hexadecimal radix. The radix is specified by a trailing O (for octal) or H for HEX. A leading `0x` may also be used for hexadecimal. Case in not important for any number or radix. Decimal is default, e.g.:

```
-AENTRY=0-0FFh-1FF
```

Did you forget the radix?

```
-AENTRY=0-0FFh-1FFh
```

(447) bad load address "*" in -A spec *(Linker)*

The load address given in a `-A` specification is invalid: it should be a valid number, in decimal, octal or hexadecimal radix. The radix is specified by a trailing O (for octal) or `H` for HEX. A leading `0x` may also be used for hexadecimal. Case in not important for any number or radix. Decimal is default, e.g.:

```
-ACODE=0h-3fffh/a000
```

Did you forget the radix?

```
-ACODE=0h-3fffh/a000h
```

(448) bad repeat count "*" in -A spec   *(Linker)*

The repeat count given in a `-A` specification is invalid, e.g.:

`-AENTRY=0-0FFhxf`

Did you forget the radix?

`-AENTRY=0-0FFhxfh`

(449) syntax error in -A spec: *   *(Linker)*

The `-A` spec is invalid. A valid -A spec should be something like:

-AROM=1000h-1FFFh

(450) psect "*" was never defined   *(Linker)*

This psect has been listed in a `-P` option, but is not defined in any module within the program.

(451) bad psect origin format in -P option   *(Linker)*

The origin format in a `-p` option is not a validly formed decimal, octal or HEX number, nor is it the name of an existing psect. A HEX number must have a trailing H, e.g.:

`-pbss=f000`

Did you forget the radix?

`-pbss=f000h`

(452) bad "+" (minimum address) format in -P option   *(Linker)*

The minimum address specification in the linker's `-p` option is badly formatted, e.g.:

`-pbss=data+f000`

Did you forget the radix?

`-pbss=data+f000h`

(453) missing number after "%" in -P option   *(Linker)*

The `%` operator in a `-p` option (for rounding boundaries) must have a number after it.

(454) link and load address can't both be set to "." in -P option   *(Linker)*

The link and load address of a psect have both been specified with a *dot* character. Only one of these addresses may be specified in this manner, e.g.:

```
-Pmypsect=1000h/.
-Pmypsect=./1000h
```

Both of these options are valid and equivalent, however the following usage is ambiguous:

`-Pmypsect=./.`

What is the link or load address of this psect?

(455) psect "*" not relocated on 0x* byte boundary   *(Linker)*

This psect is not relocated on the required boundary. Check the relocatability of the psect and correct the `-p` option. if necessary.

(456) psect "*" not loaded on 0x* boundary   *(Linker)*

This psect has a relocatability requirement that is not met by the load address given in a `-p` option. For example if a psect must be on a 4K byte boundary, you could not start it at 100H.

(459) remove failed, error: *, * *(xstrip)*

The creation of the output file failed when removing an intermediate file.

(460) rename failed, error: *, * *(xstrip)*

The creation of the output file failed when renaming an intermediate file.

(461) can't create * file "*" *(Assembler, Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(464) missing key in avmap file *(Linker)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(465) undefined symbol "*" in FNBREAK record *(Linker)*

The linker has found an undefined symbol in the FNBREAK record for a non-reentrant function. Contact HI-TECH Support if this is not handwritten assembler code.

(466) undefined symbol "*" in FNINDIR record *(Linker)*

The linker has found an undefined symbol in the FNINDIR record for a non-reentrant function. Contact HI-TECH Support if this is not handwritten assembler code.

(467) undefined symbol "*" in FNADDR record *(Linker)*

The linker has found an undefined symbol in the FNADDR record for a non-reentrant function. Contact HI-TECH Support if this is not handwritten assembler code.

(468) undefined symbol "*" in FNCALL record *(Linker)*

The linker has found an undefined symbol in the FNCALL record for a non-reentrant function. Contact HI-TECH Support if this is not handwritten assembler code.

(469) undefined symbol "*" in FNROOT record *(Linker)*

The linker has found an undefined symbol in the FNROOT record for a non-reentrant function. Contact HI-TECH Support if this is not handwritten assembler code.

(470) undefined symbol "*" in FNSIZE record *(Linker)*

The linker has found an undefined symbol in the FNSIZE record for a non-reentrant function. Contact HI-TECH Support if this is not handwritten assembler code.

(471) recursive function calls: *(Linker)*

These functions (or function) call each other recursively. One or more of these functions has statically allocated local variables (compiled stack). Either use the reentrant keyword (if supported with this compiler) or recode to avoid recursion, e.g.:

```
int test(int a)
{
   if(a == 5) {
      /* recursion may not be supported by some compilers */
      return test(a++);
   }
   return 0;
}
```

(472) non-reentrant function "*" appears in multiple call graphs: rooted at "*" and "*" *(Linker)*

This function can be called from both main-line code and interrupt code. Use the `reentrant` keyword, if this compiler supports it, or recode to avoid using local variables or parameters, or duplicate the function, e.g.:

```
void interrupt my_isr(void)
{
  scan(6);     /* scan is called from an interrupt function */
}
void process(int a)
{
  scan(a);     /* scan is also called from main-line code */
}
```

(473) function "*" is not called from specified interrupt_level *(Linker)*

The indicated function is never called from an interrupt function of the same interrupt level, e.g.:

```
#pragma interrupt_level 1
void foo(void)
{
    ...
}
#pragma interrupt_level 1
void interrupt bar(void)
{
    // this function never calls foo()
}
```

(474) no psect specified for function variable/argument allocation *(Linker)*

The `FNCONF` assembler directive which specifies to the linker information regarding the auto/parameter block was never seen. This is supplied in the standard runtime files if necessary. This error may imply that the correct run-time startup module was not linked. Ensure you have used the `FNCONF` directive if the runtime startup module is hand-written.

(475) conflicting FNCONF records *(Linker)*

The linker has seen two conflicting `FNCONF` directives. This directive should only be specified once and is included in the standard runtime startup code which is normally linked into every program.

(476) fixup overflow referencing * * (location 0x* (0x*+*), size *, value 0x*) *(Linker)*

The linker was asked to relocate (fixup) an item that would not fit back into the space after relocation. See the following error message (477) for more information.

(477) fixup overflow in expression (location 0x* (0x*+*), size *, value 0x*) *(Linker)*

Fixup is the process conducted by the linker of replacing symbolic references to variables etc, in an assembler instruction with an absolute value. This takes place after positioning the psects (program sections or blocks) into the available memory on the target device. Fixup overflow is when the value determined for a symbol is too large to fit within the allocated space within the assembler instruction. For example, if an assembler instruction has an 8-bit field to hold an address and the linker determines that the symbol that has been used to represent this address has the value 0x110, then clearly this value cannot be inserted into the instruction.

The causes for this can be many, but hand-written assembler code is always the first suspect. Badly written C code can also generate assembler that ultimately generates fixup overflow errors. Consider the following error message.

```
main.obj: 8: Fixup overflow in expression (loc 0x1FD (0x1FC+1),
    size 1, value 0x7FC)
```

This indicates that the file causing the problem was `main.obj`. This would be typically be the output of compiling `main.c` or `main.as`. This tells you the file in which you should be looking. The next number (8 in this example) is the record number in the object file that was causing the problem. If you use the DUMP utility to examine the object file, you can identify the record, however you do not normally need to do this.

The location (`loc`) of the instruction (0x1FD), the `size` (in bytes) of the field in the instruction for the value (1), and the `value` which is the actual value the symbol represents, is typically the only information needed to track down the cause of this error. Note that a size which is not a multiple of 8 bits will be rounded up to the nearest byte size, i.e. a 7 bit space in an instruction will be shown as 1 byte.

Generate an assembler list file for the appropriate module. Look for the address specified in the error message.

```
7   07FC   0E21   MOVLW 33
8   07FD   6FFC   MOVWF _foo
9   07FE   0012   RETURN
```

and to confirm, look for the symbol referenced in the assembler instruction at this address in the symbol table at the bottom of the same file.

```
Symbol Table                    Fri Aug 12 13:17:37 2004
_foo 01FC   _main 07FF
```

In this example, the instruction causing the problem takes an 8-bit offset into a bank of memory, but clearly the address 0x1FC exceeds this size. Maybe the instruction should have been written as:

```
MOVWF (_foo&0ffh)
```

which masks out the top bits of the address containing the bank information.

If the assembler instruction that caused this error was generated by the compiler, in the assembler list file look back up the file from the instruction at fault to determine which C statement has generated this instruction. You will then need to examine the C code for possible errors. incorrectly qualified pointers are an common trigger.

(478) * range check failed (location 0x* (0x*+*), value 0x* > limit 0x*) *(Linker)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(479) circular indirect definition of symbol "*" *(Linker)*

The specified symbol has been equated to an external symbol which, in turn, has been equated to the first symbol.

(480) function signatures do not match: * (*): 0x*/0x* *(Linker)*

The specified function has different signatures in different modules. This means it has been declared differently, e.g. it may have been prototyped in one module and not another. Check what declarations for the function are visible in the two modules specified and make sure they are compatible, e.g.:

```
extern int get_value(int in);
/* and in another module: */
/* this is different to the declaration */
int get_value(int in, char type)
{
```

(481) common symbol "*" psect conflict *(Linker)*

A common symbol has been defined to be in more than one psect.

(482) symbol "*" is defined more than once in "*" *(Assembler)*

This symbol has been defined in more than one place. The assembler will issue this error if a symbol is defined more than once in the same module, e.g.:

```
_next:
  MOVE r0, #55
  MOVE [r1], r0
_next:            ; oops -- choose a different name
```

The linker will issue this warning if the symbol (C or assembler) was defined multiple times in different modules. The names of the modules are given in the error message. Note that C identifiers often have an *underscore* prepended to their name after compilation.

(483) symbol "*" can't be global *(Linker)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(484) psect "*" can't be in classes "*" and "*" *(Linker)*

A psect cannot be in more than one class. This is either due to assembler modules with conflicting class= options to the PSECT directive, or use of the -C option to the linker, e.g.:

```
psect final,class=CODE
finish:
/* elsewhere: */
psect final,class=ENTRY
```

(485) unknown "with" psect referenced by psect "*" *(Linker)*

The specified psect has been placed with a psect using the psect with flag. The psect it has been placed with does not exist, e.g.:

```
psect starttext,class=CODE,with=rext
    ; was that meant to be with text?
```

(486) psect "*" selector value redefined *(Linker)*

The selector value for this psect has been defined more than once.

---

(487) psect "*" type redefined: */* *(Linker)*

This psect has had its type defined differently by different modules. This probably means you are trying to link incompatible object modules, e.g. linking 386 flat model code with 8086 real mode code.

(488) psect "*" memory space redefined: */* *(Linker)*

A global psect has been defined in two different memory spaces. Either rename one of the psects or, if they are the same psect, place them in the same memory space using the `space` psect flag, e.g.:

```
psect spdata,class=RAM,space=0
  ds 6
; elsewhere:
psect spdata,class=RAM,space=1
```

(489) psect "*" memory delta redefined: */* *(Linker)*

A global psect has been defined with two different delta values, e.g.:

```
psect final,class=CODE,delta=2
finish:
; elsewhere:
psect final,class=CODE,delta=1
```

(490) class "*" memory space redefined: */* *(Linker)*

A class has been defined in two different memory spaces. Either rename one of the classes or, if they are the same class, place them in the same memory space.

(491) can't find 0x* words for psect "*" in segment "*" *(Linker)*

One of the main tasks the linker performs is positioning the blocks (or psects) of code and data that is generated from the program into the memory available for the target device. This error indicates that the linker was unable to find an area of free memory large enough to accommodate one of the psects. The error message indicates the name of the psect that the linker was attempting to position and the segment name which is typically the name of a class which is defined with a linker `-A` option. **Section 3.7.1 "Compiler-generated Psects"** lists each compiler-generated psect and what it contains. Typically psect names which are, or include, `text` relate to program code. Names such as `bss` or `data` refer to variable blocks. This error can be due to two reasons.

First, the size of the program or the program's data has exceeded the total amount of space on the selected device. In other words, some part of your device's memory has completely filled. If this is the case, then the size of the specified psect must be reduced.

The second cause of this message is when the total amount of memory needed by the psect being positioned is sufficient, but that this memory is fragmented in such a way that the largest contiguous block is too small to accommodate the psect. The linker is unable to split psects in this situation. That is, the linker cannot place part of a psect at one location and part somewhere else. Thus, the linker must be able to find a contiguous block of memory large enough for every psect. If this is the cause of the error, then the psect must be split into smaller psects if possible.

To find out what memory is still available, generate and look in the map file, see **Section 2.7.8 "-M: Generate Map File"** for information on how to generate a map file. Search for the string UNUSED ADDRESS RANGES. Under this heading, look for the name of the segment specified in the error message. If the name is not present, then all the memory available for this psect has been allocated. If it is present, there will be

one address range specified under this segment for each free block of memory. Determine the size of each block and compare this with the number of words specified in the error message.

Psects containing code can be reduced by using all the compiler's optimizations, or restructuring the program. If a code psect must be split into two or more small psects, this requires splitting a function into two or more smaller functions (which may call each other). These functions may need to be placed in new modules.

Psects containing data may be reduced when invoking the compiler optimizations, but the effect is less dramatic. The program may need to be rewritten so that it needs less variables. **Section "If the default linker options must be changed, this can be done indirectly through the driver using the driver -L- option, see Section 2.7.7 "-L-: Adjust Linker Options Directly". If you use this option, always confirm the change appears correctly in the map file."** has information on interpreting the map file's call graph if the compiler you are using uses a compiled stack. (If the string `Call graph:` is not present in the map file, then the compiled code uses a hardware stack.) If a data psect needs to be split into smaller psects, the definitions for variables will need to be moved to new modules or more evenly spread in the existing modules. Memory allocation for `auto` variables is entirely handled by the compiler. Other than reducing the number of these variables used, the programmer has little control over their operation. This applies whether the compiled code uses a hardware or compiled stack.

For example, after receiving the message:

```
Can't find 0x34 words (0x34 withtotal) for psect text
 in segment CODE (error)
```

look in the map file for the ranges of unused memory.

```
UNUSED ADDRESS RANGES
       CODE              00000244-0000025F
                         00001000-0000102f
       RAM               00300014-00301FFB
```

In the `CODE` segment, there is 0x1c (0x25f-0x244+1) bytes of space available in one block and 0x30 available in another block. Neither of these are large enough to accommodate the psect `text` which is 0x34 bytes long. Notice, however, that the total amount of memory available is larger than 0x34 bytes.

(492) attempt to position absolute psect "*" is illegal *(Linker)*

This psect is absolute and should not have an address specified in a -P option. Either remove the `abs` psect flag, or remove the -P linker option.

(493) origin of psect "*" is defined more than once *(Linker)*

The origin of this psect is defined more than once. There is most likely more than one -p linker option specifying this psect.

(494) bad -P format "*/*" *(Linker)*

The -P option given to the linker is malformed. This option specifies placement of a psect, e.g.:

```
-Ptext=10g0h
```

Maybe you meant:

```
-Ptext=10f0h
```

(495) use of both "with=" and "INCLASS/INCLASS" allocation is illegal *(Linker)*

It is not legal to specify both the link and location of a psect as within a class, when that psect was also defined using a `with` psect flag.

(497) psect "*" exceeds max size: *h > *h *(Linker)*

The psect has more bytes in it than the maximum allowed as specified using the `size` psect flag.

(498) psect "*" exceeds address limit: *h > *h *(Linker)*

The maximum address of the psect exceeds the limit placed on it using the `limit` psect flag. Either the psect needs to be linked at a different location or there is too much code/data in the psect.

(499) undefined symbol: *(Assembler, Linker)*

The symbol following is undefined at link time. This could be due to spelling error, or failure to link an appropriate module.

(500) undefined symbols: *(Linker)*

A list of symbols follows that were undefined at link time. These errors could be due to spelling error, or failure to link an appropriate module.

(501) program entry point is defined more than once *(Linker)*

There is more than one entry point defined in the object files given the linker. End entry point is specified after the `END` directive. The runtime startup code defines the entry point, e.g.:

```
powerup:
  goto start
  END  powerup  ; end of file and define entry point
; other files that use END should not define another entry point
```

(502) incomplete * record body: length = * *(Linker)*

An object file contained a record with an illegal size. This probably means the file is truncated or not an object file. Contact HI-TECH Support with details.

(503) ident records do not match *(Linker)*

The object files passed to the linker do not have matching ident records. This means they are for different processor types.

(504) object code version is greater than *.* *(Linker)*

The object code version of an object module is higher than the highest version the linker is known to work with. Check that you are using the correct linker. Contact HI-TECH Support if the object file if you have not patched the linker.

(505) no end record found inobject file *(Linker)*

An object file did not contain an end record. This probably means the file is corrupted or not an object file. Contact HI-TECH Support if the object file was generated by the compiler.

(506) object file record too long: *+* *(Linker)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(507) unexpected end of file in object file *(Linker)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(508) relocation offset (*) out of range 0..*-*-1 *(Linker)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(509) illegal relocation size: * *(Linker)*

There is an error in the object code format read by the linker. This either means you are using a linker that is out of date, or that there is an internal error in the assembler or linker. Contact HI-TECH Support with details if the object file was created by the compiler.

(510) complex relocation not supported for -R or -L options *(Linker)*

The linker was given a -R or -L option with file that contain complex relocation.

(511) bad complex range check *(Linker)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(512) unknown complex operator 0x* *(Linker)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(513) bad complex relocation *(Linker)*

The linker has been asked to perform complex relocation that is not syntactically correct. Probably means an object file is corrupted.

(514) illegal relocation type: * *(Linker)*

An object file contained a relocation record with an illegal relocation type. This probably means the file is corrupted or not an object file. Contact HI-TECH Support with details if the object file was created by the compiler.

(515) unknown symbol type * *(Linker)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(516) text record has bad length: *-*-(*+1) < 0 *(Linker)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(520) function "*" is never called *(Linker)*

This function is never called. This may not represent a problem, but space could be saved by removing it. If you believe this function should be called, check your source code. Some assembler library routines are never called, although they are actually execute. In this case, the routines are linked in a special sequence so that program execution falls through from one routine to the next.

(521) call depth exceeded by function "*" *(Linker)*

The call graph shows that functions are nested to a depth greater than specified.

(522) library "*" is badly ordered *(Linker)*

This library is badly ordered. It will still link correctly, but it will link faster if better ordered.

(523) argument to -W option (*) illegal and ignored *(Linker)*

The argument to the linker option -w is out of range. This option controls two features. For warning levels, the range is -9 to 9. For the map file width, the range is greater than or equal to 10.

(524) unable to open list file "*": * *(Linker)*

The named list file could not be opened. The linker would be trying to fixup the list file so that it will contain absolute addresses. Ensure that an assembler list file was generated during the compilation stage. Alternatively, remove the assembler list file generation option from the link step.

(525) too many address (memory) spaces; space (*) ignored *(Linker)*

The limit to the number of address spaces (specified with the PSECT assembler directive) is currently 16.

(526) psect "*" not specified in -P option (first appears in "*") *(Linker)*

This psect was not specified in a -P or -A option to the linker. It has been linked at the end of the program, which is probably not where you wanted it.

(528) no start record; entry point defaults to zero *(Linker)*

None of the object files passed to the linker contained a start record. The start address of the program has been set to zero. This may be harmless, but it is recommended that you define a start address in your startup module by using the END directive.

(529) usage: objtohex [-Ssymfile] [object-file [HEX-file]] *(Objtohex)*

Improper usage of the command-line tool objtohex. If you are invoking objtohex directly then please refer to **Section 6.3 "Objtohex"** for more details. Otherwise this may be an internal compiler error and you should contact HI-TECH Software technical support with details.

(593) can't find 0x* words (0x* withtotal) for psect "*" in segment "*" *(Linker)*

See message (491).

(594) undefined symbol: *(Linker)*

The symbol following is undefined at link time. This could be due to spelling error, or failure to link an appropriate module.

(595) undefined symbols: *(Linker)*

A list of symbols follows that were undefined at link time. These errors could be due to spelling error, or failure to link an appropriate module.

(596) segment "*" (*-*) overlaps segment "*" (*-*) *(Linker)*

The named segments have overlapping code or data. Check the addresses being assigned by the -P linker option.

(599) No psect classes given for COFF write *(Cromwell)*

Cromwell requires that the program memory psect classes be specified to produce a COFF file. Ensure that you are using the -N option as per **Section 6.5.2 "-N"**.

(600) No chip arch given for COFF write *(Cromwell)*

Cromwell requires that the chip architecture be specified to produce a COFF file. Ensure that you are using the -P option as per Table 6-7.

(601) Unknown chip arch "*" for COFF write *(Cromwell)*

The chip architecture specified for producing a COFF file isn't recognized by Cromwell. Ensure that you are using the -P option as per **Section 6.5.1 "-Pname[,architecture]"** and that the architecture specified matches one of those in Table 6-7.

(602) null file format name *(Cromwell)*

The -I or -O option to Cromwell must specify a file format.

(603) ambiguous file format name "*" *(Cromwell)*

The input or output format specified to Cromwell is ambiguous. These formats are specified with the -i *key* and -o *key* options respectively.

(604) unknown file format name "*" *(Cromwell)*

The output format specified to CROMWELL is unknown, e.g.:

```
cromwell -m -P16F877 main.HEX main.sym -ocot
```

and output file type of cot, did you mean cof ?

(605) did not recognize format of input file *(Cromwell)*

The input file to Cromwell is required to be COD, Intel HEX, Motorola HEX, COFF, OMF51, P&E or HI-TECH.

(606) inconsistent symbol tables *(Cromwell)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(607) inconsistent line number tables *(Cromwell)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(608) bad path specification *(Cromwell)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(609) missing processor spec after -P *(Cromwell)*

The `-p` option to cromwell must specify a processor name.

(610) missing psect classes after -N *(Cromwell)*

Cromwell requires that the `-N` option be given a list of the names of psect classes.

(611) too many input files *(Cromwell)*

To many input files have been specified to be converted by `CROMWELL`.

(612) too many output files *(Cromwell)*

To many output file formats have been specified to `CROMWELL`.

(613) no output file format specified *(Cromwell)*

The output format must be specified to `CROMWELL`.

(614) no input files specified *(Cromwell)*

`CROMWELL` must have an input file to convert.

(616) option -Cbaseaddr is illegal with options -R or -L *(Linker)*

The linker option `-Cbaseaddr` cannot be used in conjunction with either the `-R` or `-L` linker options.

(618) error reading COD file data *(Cromwell)*

An error occurred reading the input COD file. Confirm the spelling and path of the file specified on the command line.

(619) I/O error reading symbol table *(Cromwell)*

The COD file has an invalid format in the specified record.

(620) filename index out of range in line number record *(Cromwell)*

The COD file has an invalid value in the specified record.

(621) error writing ELF/DWARF section "*" on "*" *(Cromwell)*

An error occurred writing the indicated section to the given file. Confirm the spelling and path of the file specified on the command line.

(622) too many type entries *(Cromwell)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(623) bad class in type hashing *(Cromwell)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(624) bad class in type compare *(Cromwell)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(625) too many files in COFF file *(Cromwell)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(626) string lookup failed in COFF: get_string() *(Cromwell)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(627) missing "*" in SDB file "*" line * column * *(Cromwell)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(629) bad storage class "*" in SDB file "*" line * column * *(Cromwell)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(630) invalid syntax for prefix list in SDB file "*" *(Cromwell)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(631) syntax error at token "*" in SDB file "*" line * column * *(Cromwell)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(632) can't handle address size (*) *(Cromwell)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(633) unknown symbol class (*) *(Cromwell)*

Cromwell has encountered a symbol class in the symbol table of a COFF, Microchip COFF, or ICOFF file which it can't identify.

(634) error dumping "*" *(Cromwell)*

Either the input file to `CROMWELL` is of an unsupported type or that file cannot be dumped to the screen.

(635) invalid HEX file "*" on line * *(Cromwell)*

The specified HEX file contains an invalid line. Contact HI-TECH Support if the HEX file was generated by the compiler.

(636) checksum error in Intel HEX file "*" on line * *(Cromwell, `HEXMATE`)*

A checksum error was found at the specified line in the specified Intel HEX file. The HEX file may be corrupt.

(637) unknown prefix "*" in SDB file "*" *(Cromwell)*

This is an internal compiler warning. Contact HI-TECH Software technical support with details.

(638) version mismatch: 0x* expected *(Cromwell)*

The input Microchip COFF file wasn't produced using Cromwell.

---

(639) zero bit width in Microchip optional header *(Cromwell)*

The optional header in the input Microchip COFF file indicates that the program or data memory spaces are zero bits wide.

(668) prefix list did not match any SDB types *(Cromwell)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(669) prefix list matched more than one SDB type *(Cromwell)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(670) bad argument to -T *(Clist)*

The argument to the -T option to specify tab size was not present or correctly formed. The option expects a decimal integer argument.

(671) argument to -T should be in range 1 to 64 *(Clist)*

The argument to the -T option to specify tab size was not in the expected range. The option expects a decimal integer argument ranging from 1 to 64 inclusive.

(673) missing filename after * option *(Objtohex)*

The indicated option requires a valid file name. Ensure that the filename argument supplied to this option exists and is spelt correctly.

(674) too many references to "*" *(Cref)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(677) set_fact_bit on pic17! *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(678) case 55 on pic17! *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(679) unknown extraspecial: * *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(680) bad format for -P option *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(681) bad common spec in -P option *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(682) this architecture is not supported by the PICC Lite compiler *(Code Generator)*

A target device other than baseline, mid-range or high end was specified. This compiler only supports devices from these architecture families.

(683) bank 1 variables are not supported by the PICC Lite compiler *(Code Generator)*

A variable with an absolute address located in bank 1 was detected. This compiler does not support code generation of variables in this bank.

(684) bank 2 and 3 variables are not supported by the PICC Lite compiler *(Code Generator)*

A variable with an absolute address located in bank 2 or 3 was detected. This compiler does not support code generation of variables in these banks.

(685) bad putwsize() *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(686) bad switch size (*) *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(687) bad pushreg "*" *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(688) bad popreg "*" *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(689) unknown predicate "*" *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(690) interrupt function requires address *(Code Generator)*

The high end PIC devices support multiple interrupts. An @ *address* is required with the interrupt definition to indicate with which vector this routine is associated, e.g.:

```
void interrupt isr(void) @ 0x10
{
  /* isr code goes here */
}
```

This construct is not required for Mid-Range PIC devices.

(691) interrupt functions not implemented for 12 bit PIC *(Code Generator)*

The 12-bit range of PIC MCU processors do not support interrupts.

(692) interrupt function "*" may only have one interrupt level *(Code Generator)*

Only one interrupt level may be associated with an `interrupt` function. Check to ensure that only one `interrupt_level` pragma has been used with the function specified. This pragma may be used more than once on main-line functions that are called from `interrupt` functions. For example:

```
#pragma interrupt_level 0
#pragma interrupt_level 1  /* which is it to be: 0 or 1? */
void interrupt isr(void)
{
```

(693) interrupt level may only be 0 (default) or 1 *(Code Generator)*

The only possible interrupt levels are 0 or 1. Check to ensure that all `interrupt_level` pragmas use these levels.

```
#pragma interrupt_level 2  /* oops -- only 0 or 1 */
void interrupt isr(void)
{
  /* isr code goes here */
}
```

(694) no interrupt strategy available *(Code Generator)*

The processor does not support saving and subsequent restoring of registers during an interrupt service routine.

(695) duplicate case label (*) *(Code Generator)*

There are two case labels with the same value in this `switch` statement, e.g.:

```
switch(in) {
case '0':  /* if this is case '0'... */
  b++;
  break;
case '0':  /* then what is this case? */
  b--;
  break;
}
```

(696) out-of-range case label (*) *(Code Generator)*

This case label is not a value that the controlling expression can yield, and thus this label will never be selected.

(697) non-constant case label *(Code Generator)*

A `case` label in this `switch` statement has a value which is not a constant.

(698) bit variables must be global or static *(Code Generator)*

A `bit` variable cannot be of type `auto`. If you require a `bit` variable with scope local to a block of code or function, qualify it `static`, e.g.:

```
bit proc(int a)
{
  bit bb;         /* oops --  this should be: static bit bb; */
  bb = (a > 66);
  return bb;
}
```

(699) no case labels in switch *(Code Generator)*

There are no case labels in this switch statement, e.g.:

```
switch(input) {
}               /* there is nothing to match the value of input */
```

(700) truncation of enumerated value *(Code Generator)*

An enumerated value larger than the maximum value supported by this compiler was detected and has been truncated, e.g.:

```
enum { ZERO, ONE, BIG=0x99999999 } test_case;
```

(701) unreasonable matching depth *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(702) regused(): bad arg to G *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(703) bad GN *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(704) bad RET_MASK *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(705) bad which (*) after I *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(706) bad which in expand() *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(707) bad SX *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(708) bad mod "+" for how = "*" *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(709) metaregister "*" can't be used directly *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(710) bad U usage *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(711) bad how in expand() *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(712) can't generate code for this expression *(Code Generator)*

This error indicates that a C expression is too difficult for the code generator to actually compile. For successful code generation, the code generator must know how to compile an expression and there must be enough resources (e.g. registers or temporary memory locations) available. Simplifying the expression, e.g. using a temporary variable to hold an intermediate result, may get around this message. Contact HI-TECH Support with details of this message.

This error may also be issued if the code being compiled is in some way unusual. For example code which writes to a const-qualified object is illegal and will result in warning messages, but the code generator may unsuccessfully try to produce code to perform the write.

(713) bad initialization list *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(714) bad intermediate code *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(715) bad pragma "*" *(Code Generator)*

The code generator has been passed a `pragma` directive that it does not understand. This implies that the pragma you have used is a HI-TECH specific pragma, but the specific compiler you are using has not implemented this pragma.

(716) bad argument to -M option "*" *(Code Generator)*

The code generator has been passed a `-M` option that it does not understand. This should not happen if it is being invoked by a standard compiler driver.

**(718) incompatible intermediate code version; should be *.* ** *(Code Generator)*

The intermediate code file produced by P1 is not the correct version for use with this code generator. This is either that incompatible versions of one or more compilers have been installed in the same directory, or a temporary file error has occurred leading to corruption of a temporary file. Check the setting of the TEMP environment variable. If it refers to a long path name, change it to something shorter. Contact HI-TECH Support with details if required.

(720) multiple free: * *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(721) element count must be constant expression *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(722) bad variable syntax in intermediate code *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(723) function definitions nested too deep *(Code Generator)*

This error is unlikely to happen with C code, since C cannot have nested functions! Contact HI-TECH Support with details.

(724) bad op (*) in revlog() *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(726) bad op "*" in uconval() *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(727) bad op "*" in bconfloat() *(Code Generator)*

This is an internal code generator error. Contact HI-TECH technical support with details.

(728) bad op "*" in confloat() *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(729) bad op "*" in conval() *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(730) bad op "*" *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(731) expression error with reserved word *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(732) initialization of bit types is illegal *(Code Generator)*

Variables of type `bit` cannot be initialized, e.g.:

```
bit b1 = 1; /* oops!
             b1 must be assigned after its definition */
```

(733) bad string "*" in pragma "psect" *(Code Generator)*

The code generator has been passed a `pragma psect` directive that has a badly formed string, e.g.:

```
#pragma psect text  /* redirect text psect into what? */
```

Maybe you meant something like:

```
#pragma psect text=special_text
```

(734) too many "psect" pragmas *(Code Generator)*

Too many `#pragma psect` directives have been used.

---

(735) bad string "*" in pragma "stack_size" *(Code Generator)*

The argument to the stack_size pragma is malformed. This pragma must be followed by a number representing the maximum allowed stack size.

(737) unknown argument "*" to pragma "switch" *(Code Generator)*

The `#pragma switch` directive has been used with an invalid switch code generation method. Possible arguments are: `auto`, `simple` and `direct`.

(739) error closing output file *(Code Generator)*

The compiler detected an error when closing a file. Contact HI-TECH Support with details.

(740) zero dimension array is illegal *(Code Generator)*

The code generator has been passed a declaration that results in an array having a zero dimension.

(741) bitfield too large (* bits) *(Code Generator)*

The maximum number of bits in a bit field is the same as the number of bits in an `int`, e.g. assuming an `int` is 16 bits wide:

```
struct {
  unsigned flag  : 1;
  unsigned value : 12;
  unsigned cont  : 6;   /* oops -- that's a total of 19 bits */
} object;
```

(742) function "*" argument evaluation overlapped *(Linker)*

A function call involves arguments which overlap between two functions. This could occur with a call like:

```
void fn1(void)
{
  fn3( 7, fn2(3), fn2(9));    /* Offending call */
}
char fn2(char fred)
{
  return fred + fn3(5,1,0);
}
char fn3(char one, char two, char three)
{
  return one+two+three;
}
```

where `fn1` is calling `fn3`, and two arguments are evaluated by calling `fn2`, which in turn calls `fn3`. The program structure should be modified to prevent this type of call sequence.

(743) divide by zero *(Code Generator)*

An expression involving a division by zero has been detected in your code.

(744) static object "*" has zero size *(Code Generator)*

A static object has been declared, but has a size of zero.

(745) nodecount = * *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(746) object "*" qualified const, but not initialized *(Code Generator)*

An object has been qualified as `const`, but there is no initial value supplied at the definition. As this object cannot be written by the C program, this may imply the initial value was accidently omitted.

(747) unrecognized option "*" to -Z *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(748) variable "*" may be used before set *(Code Generator)*

This variable may be used before it has been assigned a value. Since it is an `auto` variable, this will result in it having a random value, e.g.:

```
void main(void)
{
  int a;
  if(a)         /* oops -- a has never been assigned a value */
    process();
}
```

(749) unknown register name "*" used with pragma *(Linker)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(750) constant operand to || or && *(Code Generator)*

One operand to the logical operators `||` or `&&` is a constant. Check the expression for missing or badly placed parentheses. This message may also occur if the global optimizer is enabled and one of the operands is an `auto` or `static` local variable whose value has been tracked by the code generator, e.g.:

```
{
int a;
a = 6;
if(a || b)  /* a is 6, therefore this is always true */
    b++;
```

(751) arithmetic overflow in constant expression   *(Code Generator)*

A constant expression has been evaluated by the code generator that has resulted in a value that is too big for the type of the expression. The most common code to trigger this warning is assignments to `signed` data types. For example:

```
signed char c;
c = 0xFF;
```

As a `signed` 8-bit quantity, `c` can only be assigned values -128 to 127. The constant is equal to 255 and is outside this range. If you mean to set all bits in this variable, then use either of:

```
c = ~0x0;
c = -1;
```

which will set all the bits in the variable regardless of the size of the variable and without warning.

This warning can also be triggered by intermediate values overflowing. For example:

```
unsigned int i;  /* assume ints are 16 bits wide */
i = 240 * 137;   /* this should be okay, right? */
```

A quick check with your calculator reveals that 240 * 137 is 32880 which can easily be stored in an `unsigned int`, but a warning is produced. Why? Because 240 and 137 and both `signed int` values. Therefore the result of the multiplication must also be a `signed int` value, but a `signed int` cannot hold the value 32880. (Both operands are constant values so the code generator can evaluate this expression at compile time, but it must do so following all the ANSI C rules.) The following code forces the multiplication to be performed with an `unsigned` result:

```
i = 240u * 137;  /* force at least one operand
                    to be unsigned */
```

(752) conversion to shorter data type   *(Code Generator)*

Truncation may occur in this expression as the lvalue is of shorter type than the rvalue, e.g.:

```
char a;
int b, c;
a = b + c;  /* int to char conversion
               may result in truncation */
```

(753) undefined shift (* bits)   *(Code Generator)*

An attempt has been made to shift a value by a number of bits equal to or greater than the number of bits in the data type. This will produce an undefined result on many processors. This is non-portable code and is flagged as having undefined results by the C Standard, e.g.:

```
int input;
input <<= 33;  /* oops -- that shifts the entire value out */
```

(754) bitfield comparison out of range   *(Code Generator)*

This is the result of comparing a bitfield with a value when the value is out of range of the bitfield. For example, comparing a 2-bit bitfield to the value 5 will never be true as a 2-bit bitfield has a range from 0 to 3, e.g.:

```
struct {
  unsigned mask : 2;  /* mask can hold values 0 to 3 */
} value;
int compare(void)
{
  return (value.mask == 6);  /* test can
}
```

(755) divide by zero   *(Code Generator)*

A constant expression that was being evaluated involved a division by zero, e.g.:

```
a /= 0;  /* divide by 0: was this what you were intending */
```

(757) constant conditional branch   *(Code Generator)*

A conditional branch (generated by an `if`, `for`, `while` statement etc.) always follows the same path. This will be some sort of comparison involving a variable and a constant expression. For the code generator to issue this message, the variable must have local scope (either `auto` or `static` local) and the global optimizer must be enabled, possibly at higher level than 1, and the warning level threshold may need to be lower than the default level of 0.

The global optimizer keeps track of the contents of local variables for as long as is possible during a function. For C code that compares these variables to constants, the result of the comparison can be deduced at compile time and the output code hard coded to avoid the comparison, e.g.:

```
{
  int a, b;
  a = 5;
  /* this can never be false;
     always perform the true statement */
  if(a == 4)
    b = 6;
```

will produce code that sets `a` to 5, then immediately sets `b` to 6. No code will be produced for the comparison `if(a == 4)`. If a was a global variable, it may be that other functions (particularly interrupt functions) may modify it and so tracking the variable cannot be performed.

This warning may indicate more than an optimization made by the compiler. It may indicate an expression with missing or badly placed parentheses, causing the evaluation to yield a value different to what you expected.

This warning may also be issued because you have written something like `while(1)`. To produce an infinite loop, use `for(;;)`.

A similar situation arises with for loops, e.g.:

```
{
  int a, b;
  /* this loop must iterate at least once */
  for(a=0; a!=10; a++)
    b = func(a);
```

In this case the code generator can again pick up that a is assigned the value 0, then immediately checked to see if it is equal to 10. Because a is modified during the `for` loop, the comparison code cannot be removed, but the code generator will adjust the code so that the comparison is not performed on the first pass of the loop; only on the subsequent passes. This may not reduce code size, but it will speed program execution.

(758) constant conditional branch: possible use of "=" instead of "==" *(Code Generator)*

There is an expression inside an `if` or other conditional construct, where a constant is being assigned to a variable. This may mean you have inadvertently used an assignment = instead of a compare ==, e.g.:

```
int a, b;
/* this can never be false;
   always perform the true statement */
if(a = 4)
  b = 6;
```

will assign the value 4 to a, then , as the value of the assignment is always true, the comparison can be omitted and the assignment to b always made. Did you mean:

```
/* this can never be false;
   always perform the true statement */
if(a == 4)
   b = 6;
```

which checks to see if a is equal to 4.

(759) expression generates no code *(Code Generator)*

This expression generates no output code. Check for things like leaving off the parentheses in a function call, e.g.:

```
int fred;
fred;     /* this is valid, but has no effect at all */
```

Some devices require that special function register need to be read to clear hardware flags. To accommodate this, in some instances the code generator *does* produce code for a statement which only consists of a variable ID. This may happen for variables which are qualified as `volatile`. Typically the output code will read the variable, but not do anything with the value read.

(760) portion of expression has no effect *(Code Generator)*

Part of this expression has no side effects, and no effect on the value of the expression, e.g.:

```
int a, b, c;
a = b,c;  /* "b" has no effect,
             was that meant to be a comma? */
```

(761) sizeof yields 0 *(Code Generator)*

The code generator has taken the size of an object and found it to be zero. This almost certainly indicates an error in your declaration of a pointer, e.g. you may have declared a pointer to a zero length array. In general, pointers to arrays are of little use. If you require a pointer to an array of objects of unknown length, you only need a pointer to a single object that can then be indexed or incremented.

(762) constant truncated when assigned to bitfield *(Code Generator)*

A constant value is too large for a bitfield structure member to which it is being assigned, e.g.

```
struct INPUT {
  unsigned a : 3;
  unsigned b : 5;
} input_grp;
input_grp.a = 0x12;
/* 12h cannot fit into a 3-bit wide object */
```

(763) constant left operand to "? :" operator *(Code Generator)*

The left operand to a conditional operator ? is constant, thus the result of the tertiary operator ?: will always be the same, e.g.:

```
a = 8 ? b : c;  /* this is the same as saying a = b; */
```

(764) mismatched comparison *(Code Generator)*

A comparison is being made between a variable or expression and a constant value which is not in the range of possible values for that expression, e.g.:

```
unsigned char c;
if(c > 300)      /* oops -- how can this be true? */
  close();
```

(765) degenerate unsigned comparison *(Code Generator)*

There is a comparison of an `unsigned` value with zero, which will always be true or false, e.g.:

```
unsigned char c;
if(c >= 0)
```

will always be true, because an `unsigned` value can never be less than zero.

(766) degenerate signed comparison *(Code Generator)*

There is a comparison of a `signed` value with the most negative value possible for this type, such that the comparison will always be true or false, e.g.:

```
char c;
if(c >= -128)
```

will always be true, because an 8 bit signed char has a maximum negative value of -128.

(767) constant truncated to bitfield width *(Code Generator)*

A constant value is too large for a bitfield structure member on which it is operating, e.g.

```
struct INPUT {
  unsigned a : 3;
  unsigned b : 5;
} input_grp;
input_grp.a |= 0x13;
/* 13h to large for 3-bit wide object */
```

(768) constant relational expression *(Code Generator)*

There is a relational expression that will always be true or false. This may be because e.g. you are comparing an `unsigned` number with a negative value, or comparing a variable with a value greater than the largest number it can represent, e.g.:

```
unsigned int a;
if(a == -10)    /* if a is unsigned, how can it be -10? */
  b = 9;
```

(769) no space for macro definition *(Assembler)*

The assembler has run out of memory.

(772) include files nested too deep *(Assembler)*

Macro expansions and include file handling have filled up the assembler's internal stack. The maximum number of open macros and include files is 30.

(773) macro expansions nested too deep *(Assembler)*

Macro expansions in the assembler are nested too deep. The limit is 30 macros and include files nested at one time.

(774) too many macro parameters *(Assembler)*

There are too many macro parameters on this macro definition.

(776) can't allocate space for object "*" (offs: *) *(Assembler)*

The assembler has run out of memory.

(777) can't allocate space for opnd structure within object "*", (offs: *) *(Assembler)*

The assembler has run out of memory.

(780) too many psects defined *(Assembler)*

There are too many psects defined! Boy, what a program!

(781) can't enter abs psect *(Assembler)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(782) REMSYM error *(Assembler)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(783) "with" psects are cyclic *(Assembler)*

If Psect A is to be placed "with" Psect B, and Psect B is to be placed "with" Psect A, there is no hierarchy. The `with` flag is an attribute of a psect and indicates that this psect must be placed in the same memory page as the specified psect.

Remove a `with` flag from one of the psect declarations. Such an assembler declaration may look like:

```
psect my_text,local,class=CODE,with=basecode
```

which will define a psect called `my_text` and place this in the same page as the psect `basecode`.

(784) overfreed *(Assembler)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(785) too many temporary labels *(Assembler)*

There are too many temporary labels in this assembler file. The assembler allows a maximum of 2000 temporary labels.

(787) can't handle "v_rtype" of * in copyexpr *(Assembler)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(788) invalid character "*" in number *(Assembler)*

A number contained a character that was not part of the range 0-9 or 0-F.

(790) end of file inside conditional *(Assembler)*

END-of-FILE was encountered while scanning for an "endif" to match a previous "if".

(793) unterminated macro argument *(Assembler)*

An argument to a macro is not terminated. Note that angle brackets ("< >") are used to quote macro arguments.

(794) invalid number syntax *(Assembler)*

The syntax of a number is invalid. This can be, e.g. use of 8 or 9 in an octal number, or other malformed numbers.

(796) use of LOCAL outside macros is illegal *(Assembler)*

The LOCAL directive is only legal inside macros. It defines local labels that will be unique for each invocation of the macro.

(797) syntax error in LOCAL argument *(Assembler)*

A symbol defined using the LOCAL assembler directive in an assembler macro is syntactically incorrect. Ensure that all symbols and all other assembler identifiers conform with the assembly language of the target device.

(798) macro argument may not appear after LOCAL *(Assembler)*

The list of labels after the directive LOCAL may not include any of the formal parameters to the macro, e.g.:

```
mmm MACRO a1
    MOVE   r0, #a1
    LOCAL  a1        ; oops --
; the macro parameter cannot be used with local
ENDM
```

(799) REPT argument must be >= 0 *(Assembler)*

The argument to a REPT directive must be greater than zero, e.g.:

```
REPT -2              ; -2 copies of this code? */
  MOVE    r0, [r1]++
ENDM
```

(800) undefined symbol "*" _(Assembler)_

The named symbol is not defined in this module, and has not been specified GLOBAL.

(801) range check too complex _(Assembler)_

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(802) invalid address after END directive _(Assembler)_

The start address of the program which is specified after the assembler END directive must be a label in the current file.

(803) undefined temporary label _(Assembler)_

A temporary label has been referenced that is not defined. Note that a temporary label must have a number >= 0.

(804) write error on object file _(Assembler)_

The assembler failed to write to an object file. This may be an internal compiler error. Contact HI-TECH Software technical support with details.

(806) attempted to get an undefined object (*) _(Assembler)_

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(807) attempted to set an undefined object (*) _(Assembler)_

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(808) bad size in add_reloc() _(Assembler)_

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(809) unknown addressing mode (*) _(Assembler)_

An unknown addressing mode was used in the assembly file.

(811) "cnt" too large (*) in display() _(Assembler)_

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(814) processor type not defined _(Assembler)_

The processor must be defined either from the command line (eg. -16c84), via the PROCESSOR assembler directive, or via the LIST assembler directive.

(815) syntax error in chipinfo file at line * _(Assembler)_

The chipinfo file contains non-standard syntax at the specified line.

(816) duplicate ARCH specification in chipinfo file "*" at line * _(Assembler, Driver)_

The chipinfo file has a processor section with multiple ARCH values. Only one ARCH value is allowed. If you have not manually edited the chip info file, contact HI-TECH Support with details.

(817) unknown architecture in chipinfo file at line *   *(Assembler, Driver)*

An chip architecture (family) that is unknown was encountered when reading the chip INI file.

(818) duplicate BANKS for "*" in chipinfo file at line *   *(Assembler)*

The chipinfo file has a processor section with multiple BANKS values. Only one BANKS value is allowed. If you have not manually edited the chip info file, contact HI-TECH Support with details.

(819) duplicate ZEROREG for "*" in chipinfo file at line *   *(Assembler)*

The chipinfo file has a processor section with multiple ZEROREG values. Only one ZEROREG value is allowed. If you have not manually edited the chip info file, contact HI-TECH Support with details.

(820) duplicate SPAREBIT for "*" in chipinfo file at line *   *(Assembler)*

The chipinfo file has a processor section with multiple SPAREBIT values. Only one SPAREBIT value is allowed. If you have not manually edited the chip info file, contact HI-TECH Support with details.

(821) duplicate INTSAVE for "*" in chipinfo file at line *   *(Assembler)*

The chipinfo file has a processor section with multiple INTSAVE values. Only one INTSAVE value is allowed. If you have not manually edited the chip info file, contact HI-TECH Support with details.

(822) duplicate ROMSIZE for "*" in chipinfo file at line *   *(Assembler)*

The chipinfo file has a processor section with multiple ROMSIZE values. Only one ROMSIZE value is allowed. If you have not manually edited the chip info file, contact HI-TECH Support with details.

(823) duplicate START for "*" in chipinfo file at line *   *(Assembler)*

The chipinfo file has a processor section with multiple START values. Only one START value is allowed. If you have not manually edited the chip info file, contact HI-TECH Support with details.

(824) duplicate LIB for "*" in chipinfo file at line *   *(Assembler)*

The chipinfo file has a processor section with multiple LIB values. Only one LIB value is allowed. If you have not manually edited the chip info file, contact HI-TECH Support with details.

(825) too many RAMBANK lines in chipinfo file for "*"   *(Assembler)*

The chipinfo file contains a processor section with too many RAMBANK fields. Reduce the number of values.

(826) inverted ram bank in chipinfo file at line *   *(Assembler, Driver)*

The second HEX number specified in the RAM field in the chipinfo file must be greater in value than the first.

(827) too many COMMON lines in chipinfo file for "*"   *(Assembler)*

There are too many lines specifying common (access bank) memory in the chip configuration file.

(828) inverted common bank in chipinfo file at line *   *(Assembler, Driver)*

The second HEX number specified in the COMMON field in the chipinfo file must be greater in value than the first. Contact HI-TECH Support if you have not modified the chipinfo INI file.

(829) unrecognized line in chipinfo file at line *   *(Assembler)*

The chipinfo file contains a processor section with an unrecognized line. Contact HI-TECH Support if the INI has not been edited.

(830) missing ARCH specification for "*" in chipinfo file   *(Assembler)*

The chipinfo file has a processor section without an ARCH values. The architecture of the processor must be specified. Contact HI-TECH Support if the chipinfo file has not been modified.

(832) empty chip info file "*"   *(Assembler)*

The chipinfo file contains no data. If you have not manually edited the chip info file, contact HI-TECH Support with details.

(833) no valid entries in chipinfo file   *(Assembler)*

The chipinfo file contains no valid processor descriptions.

(834) page width must be >= 60   *(Assembler)*

The listing page width must be at least 60 characters. Any less will not allow a properly formatted listing to be produced, e.g.:

```
LIST C=10  ; the page width will need to be wider than this
```

(835) form length must be >= 15   *(Assembler)*

The form length specified using the -F *length* option must be at least 15 lines. Setting this length to zero is allowed and turns off paging altogether. The default value is zero (pageless).

(836) no file arguments   *(Assembler)*

The assembler has been invoked without any file arguments. It cannot assemble anything.

(839) relocation too complex   *(Assembler)*

The complex relocation in this expression is too big to be inserted into the object file.

(840) phase error   *(Assembler)*

The assembler has calculated a different value for a symbol on two different passes. This is probably due to bizarre use of macros or conditional assembly.

(841) bad source/destination for movfp/movpf instruction   *(Assembler)*

The absolute address specified with the MOVFP/MOVPF instruction is too large.

(842) bad bit number   *(Assembler)*

A bit number must be an absolute expression in the range 0-7.

(843) a macro name can't also be an EQU/SET symbol   *(Assembler)*

An EQU or SET symbol has been found with the same name as a macro. This is not
allowed. For example:

```
getval MACRO
  MOV  r0, r1
ENDM
getval EQU 55h  ; oops -- choose a different name to the macro
```

(844) lexical error   *(Assembler)*

An unrecognized character or token has been seen in the input.

(845) symbol "*" defined more than once   *(Assembler)*

This symbol has been defined in more than one place. The assembler will issue this
error if a symbol is defined more than once in the same module, e.g.:

```
_next:
    MOVE   r0, #55
    MOVE   [r1], r0
_next:            ; oops -- choose a different name
```

The linker will issue this warning if the symbol (C or assembler) was defined multiple
times in different modules. The names of the modules are given in the error message.
Note that C identifiers often have an *underscore* prepended to their name after
compilation.

(846) relocation error   *(Assembler)*

It is not possible to add together two relocatable quantities. A constant may be added
to a relocatable value, and two relocatable addresses in the same psect may be sub-
tracted. An absolute value must be used in various places where the assembler must
know a value at assembly time.

(847) operand error   *(Assembler)*

The operand to this opcode is invalid. Check your assembler reference manual for the
proper form of operands for this instruction.

(848) symbol has been declared EXTERN   *(Assembler)*

An assembly label uses the same name as a symbol that has already been declared
as EXTERN.

(849) illegal instruction for this processor   *(Assembler)*

The instruction is not supported by this processor.

(850) PAGESEL not usable with this processor   *(Assembler)*

The PAGESEL pseudo-instruction is not usable with the device selected.

(851) illegal destination   *(Assembler)*

The destination (either ,f or ,w) is not correct for this instruction.

(852) radix must be from 2 - 16   *(Assembler)*

The radix specified using the RADIX assembler directive must be in the range from 2
(binary) to 16 (hexadecimal).

---

(853) invalid size for FNSIZE directive *(Assembler)*

The assembler FNSIZE assembler directive arguments must be positive constants.

(855) ORG argument must be a positive constant *(Assembler)*

An argument to the ORG assembler directive must be a positive constant or a symbol which has been equated to a positive constant, e.g.:

```
ORG -10  /* this must a positive offset to the current psect */
```

(856) ALIGN argument must be a positive constant *(Assembler)*

The align assembler directive requires a non-zero positive integer argument.

(857) psect may not be local and global *(Linker)*

A local psect may not have the same name as a global psect, e.g.:

```
psect text,class=CODE      ; text is implicitly global
   MOVE  r0, r1
; elsewhere:
psect text,local,class=CODE
   MOVE  r2, r4
```

The global flag is the default for a psect if its scope is not explicitly stated.

(859) argument to C option must specify a positive constant *(Assembler)*

The parameter to the LIST assembler control's C= option (which sets the column width of the listing output) must be a positive decimal constant number, e.g.:

```
LIST C=a0h  ; constant must be decimal and positive,
             try: LIST C=80
```

(860) page width must be >= 49 *(Assembler)*

The page width suboption to the LIST assembler directive must specify a with of at least 49.

(861) argument to N option must specify a positive constant *(Assembler)*

The parameter to the LIST assembler control's N option (which sets the page length for the listing output) must be a positive constant number, e.g.:

```
LIST N=-3  ; page length must be positive
```

(862) symbol is not external *(Assembler)*

A symbol has been declared as EXTRN but is also defined in the current module.

(863) symbol can't be both extern and public *(Assembler)*

If the symbol is declared as extern, it is to be imported. If it is declared as public, it is to be exported from the current module. It is not possible for a symbol to be both.

(864) argument to "size" psect flag must specify a positive constant *(Assembler)*

The parameter to the PSECT assembler directive's size option must be a positive constant number, e.g.:

```
PSECT text,class=CODE,size=-200  ; a negative size?
```

(865) psect flag "size" redefined *(Assembler)*

The `size` flag to the `PSECT` assembler directive is different from a previous `PSECT` directive, e.g.:

```
psect spdata,class=RAM,size=400
; elsewhere:
psect spdata,class=RAM,size=500
```

(866) argument to "reloc" psect flag must specify a positive constant *(Assembler)*

The parameter to the `PSECT` assembler directive's `reloc` option must be a positive constant number, e.g.:

```
psect test,class=CODE,reloc=-4  ; the reloc must be positive
```

(867) psect flag "reloc" redefined *(Assembler)*

The `reloc` flag to the `PSECT` assembler directive is different from a previous `PSECT` directive, e.g.:

```
psect spdata,class=RAM,reloc=4
; elsewhere:
psect spdata,class=RAM,reloc=8
```

(868) argument to "delta" psect flag must specify a positive constant *(Assembler)*

The parameter to the `PSECT` assembler directive's `DELTA` option must be a positive constant number, e.g.:

```
PSECT text,class=CODE,delta=-2  ; negative delta value doesn't make
sense
```

(869) psect flag "delta" redefined *(Assembler)*

The 'DELTA' option of a psect has been redefined more than once in the same module.

(870) argument to "pad" psect flag must specify a positive constant *(Assembler)*

The parameter to the PSECT assembler directive's 'PAD' option must be a non-zero positive integer.

(871) argument to "space" psect flag must specify a positive constant *(Assembler)*

The parameter to the PSECT assembler directive's `space` option must be a positive constant number, e.g.:

```
PSECT text,class=CODE,space=-1  ; space values start at zero
```

(872) psect flag "space" redefined *(Assembler)*

The `space` flag to the `PSECT` assembler directive is different from a previous `PSECT` directive, e.g.:

```
psect spdata,class=RAM,space=0
; elsewhere:
psect spdata,class=RAM,space=1
```

(873) a psect may only be in one class *(Assembler)*

You cannot assign a psect to more than one class. The psect was defined differently at this point than when it was defined elsewhere. A psect's class is specified via a flag as in the following:

```
psect text,class=CODE
```

Look for other psect definitions that specify a different class name.

(874) a psect may only have one "with" option *(Assembler)*

A psect can only be placed `with` one other psect. A psect's with option is specified via a flag as in the following:

```
psect bss,with=data
```

Look for other psect definitions that specify a different `with` psect name.

(875) bad character constant in expression *(Assembler)*

The character constant was expected to consist of only one character, but was found to be greater than one character or none at all. An assembler specific example:

```
MOV  r0, #'12'   ; '12' specifies two characters
```

(876) syntax error *(Assembler)*

A syntax error has been detected. This could be caused a number of things.

(877) yacc stack overflow *(Assembler)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(878) -S option used: "*" ignored *(Driver)*

The indicated assembly file has been supplied to the driver in conjunction with the `-S` option. The driver really has nothing to do since the file is already an assembly file.

(880) invalid number of parameters. Use "* –HELP" for help *(Driver)*

Improper command-line usage of the of the compiler's driver.

(881) setup succeeded *(Driver)*

The compiler has been successfully setup using the `--setup` driver option.

(883) setup failed *(Driver)*

The compiler was not successfully setup using the `--setup` driver option. Ensure that the directory argument to this option is spelt correctly, is syntactically correct for your host operating system and it exists.

(884) please ensure you have write permissions to the configuration file *(Driver)*

The compiler was not successfully setup using the `--setup` driver option because the driver was unable to access the XML configuration file. Ensure that you have write permission to this file. The driver will search the following configuration files in order:

- the file specified by the environment variable `HTC_XML`
- the file `/etc/htsoft.xml` if the directory '`/etc`' is writable and there is no `.htsoft.xml` file in your home directory
- the file `.htsoft.xml` file in your home directory

If none of the files can be located then the above error will occur.

(889) this * compiler has expired *(Driver)*

The demo period for this compiler has concluded.

(890) contact HI-TECH Software to purchase and re-activate this compiler *(Driver)*

The evaluation period of this demo installation of the compiler has expired. You will need to purchase the compiler to re-activate it. If however you sincerely believe the evaluation period has ended prematurely please contact HI-TECH technical support.

(891) can't open psect usage map file "*": * *(Driver)*

The driver was unable to open the indicated file. The psect usage map file is generated by the driver when the driver option `--summary=file` is used. Ensure that the file is not open in another application.

(892) can't open memory usage map file "*": * *(Driver)*

The driver was unable to open the indicated file. The memory usage map file is generated by the driver when the driver option `--summary=file` is used. Ensure that the file is not open in another application.

(893) can't open HEX usage map file "*": * *(Driver)*

The driver was unable to open the indicated file. The HEX usage map file is generated by the driver when the driver option `--summary=file` is used. Ensure that the file is not open in another application.

(894) unknown source file type "*" *(Driver)*

The extension of the indicated input file could not be determined. Only files with the extensions `as`, `c`, `obj`, `usb`, `p1`, `lib` or `HEX` are identified by the driver.

(895) can't request and specify options in the one command *(Driver)*

The usage of the driver options `--getoption` and `--setoption` is mutually exclusive.

(896) no memory ranges specified for data space *(Driver)*

No on-chip or external memory ranges have been specified for the data space memory for the device specified.

(897) no memory ranges specified for program space *(Driver)*

No on-chip or external memory ranges have been specified for the program space memory for the device specified.

(899) can't open option file "*" for application "*": * *(Driver)*

An option file specified by a `--getoption` or `--setoption` driver option could not be opened. If you are using the `--setoption` option ensure that the name of the file is spelt correctly and that it exists. If you are using the `--getoption` option ensure that this file can be created at the given location or that it is not in use by any other application.

(900) exec failed: * *(Driver)*

The subcomponent listed failed to execute. Does the file exist? Try re-installing the compiler.

(902) no chip name specified; use "* –CHIPINFO" to see available chip names *(Driver)*

The driver was invoked without selecting what chip to build for. Running the driver with the –CHIPINFO option will display a list of all chips that could be selected to build for.

(904) illegal format specified in "*" option *(Driver)*

The usage of this option was incorrect. Confirm correct usage with –HELP or refer to the part of the manual that discusses this option.

(905) illegal application specified in "*" option *(Driver)*

The application given to this option is not understood or does not belong to the compiler.

(907) unknown memory space tag "*" in "*" option specification *(Driver)*

A parameter to this memory option was a string but did not match any valid *tags*. Refer to the section of this manual that describes this option to see what tags (if any) are valid for this device.

(908) exit status = * *(Driver)*

One of the subcomponents being executed encountered a problem and returned an error code. Other messages should have been reported by the subcomponent to explain the problem that was encountered.

(913) "*" option may cause compiler errors in some standard header files *(Driver)*

Using this option will invalidate some of the qualifiers used in the standard header files resulting in errors. This issue and its solution are detailed in the section of this manual that specifically discusses this option.

(915) no room for arguments *(Preprocessor, Parser, Code Generator, Linker, Objtohex)*

The code generator could not allocate any more memory.

(917) argument too long *(Preprocessor, Parser)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(918) *: no match *(Preprocessor, Parser)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(919) * in chipinfo file "*" at line * *(Driver)*

The specified parameter in the chip configuration file is illegal.

(920) empty chipinfo file *(Driver, Assembler)*

The chip configuration file was able to be opened but it was empty. Try re-installing the compiler.

(922) chip "*" not present in chipinfo file "*" *(Driver)*

The chip selected does not appear in the compiler's chip configuration file. You may need to contact HI-TECH Software to see if support for this device is available or upgrade the version of your compiler.

(923) unknown suboption "*" *(Driver)*

This option can take suboptions, but this suboption is not understood. This may just be a simple spelling error. If not, –HELP to look up what suboptions are permitted here.

(924) missing argument to "*" option *(Driver)*

This option expects more data but none was given. Check the usage of this option.

(925) extraneous argument to "*" option *(Driver)*

This option does not accept additional data, yet additional data was given. Check the usage of this option.

(926) duplicate "*" option *(Driver)*

This option can only appear once, but appeared more than once.

(928) bad "*" option value *(Driver, Assembler)*

The indicated option was expecting a valid hexadecimal integer argument.

(929) bad "*" option ranges *(Driver)*

This option was expecting a parameter in a range format (*start_of_range-end_of_range*), but the parameter did not conform to this syntax.

(930) bad "*" option specification *(Driver)*

The parameters to this option were not specified correctly. Run the driver with –HELP or refer to the driver's chapter in this manual to verify the correct usage of this option.

(931) command file not specified *(Driver)*

Command file to this application, expected to be found after '@' or '<' on the command line was not found.

(939) no file arguments *(Driver)*

The driver has been invoked with no input files listed on its command line. If you are getting this message while building through a third party IDE, perhaps the IDE could not verify the source files to compile or object files to link and withheld them from the command line.

(940) *-bit checksum * placed at * *(Objtohex)*

Presenting the result of the requested checksum calculation.

(941) bad "*" assignment; USAGE: ** *(Hexmate)*

An option to HEXMATE was incorrectly used or incomplete. Follow the usage supplied by the message and ensure that the option has been formed correctly and completely.

(942) unexpected character on line * of file "*" *(Hexmate)*

File contains a character that was not valid for this type of file, the file may be corrupt. For example, an Intel HEX file is expected to contain only ASCII representations of hexadecimal digits, colons (:) and line formatting. The presence of any other characters will result in this error.

(944) data conflict at address *h between * and * *(Hexmate)*

Sources to Hexmate request differing data to be stored to the same address. To force one data source to override the other, use the '+' specifier. If the two named sources of conflict are the same source, then the source may contain an error.

(945) checksum range (*h to *h) contained an indeterminate value *(Hexmate)*

The range for this checksum calculation contained a value that could not be resolved. This can happen if the checksum result was to be stored within the address range of the checksum calculation.

(948) checksum result width must be between 1 and 4 bytes *(Hexmate)*

The requested checksum byte size is illegal. Checksum results must be within 1 to 4 bytes wide. Check the parameters to the -CKSUM option.

(949) start of checksum range must be less than end of range *(Hexmate)*

The -CKSUM option has been given a range where the start is greater than the end. The parameters may be incomplete or entered in the wrong order.

(951) start of fill range must be less than end of range *(Hexmate)*

The -FILL option has been given a range where the start is greater than the end. The parameters may be incomplete or entered in the wrong order.

(953) unknown -HELP sub-option: * *(Hexmate)*

Invalid sub-option passed to -HELP. Check the spelling of the sub-option or use -HELP with no sub-option to list all options.

(956) -SERIAL value must be between 1 and * bytes long *(Hexmate)*

The serial number being stored was out of range. Ensure that the serial number can be stored in the number of bytes permissible by this option.

(958) too many input files specified; * file maximum *(Hexmate)*

Too many file arguments have been used. Try merging these files in several stages rather than in one command.

(960) unexpected record type (*) on line * of "*" *(Hexmate)*

Intel HEX file contained an invalid record type. Consult the Intel HEX format specification for valid record types.

(962) forced data conflict at address *h between * and * *(Hexmate)*

Sources to HEXMATE force differing data to be stored to the same address. More than one source using the '+' specifier store data at the same address. The actual data stored there may not be what you expect.

(963) checksum range includes voids or unspecified memory locations *(Hexmate)*

Checksum range had gaps in data content. The runtime calculated checksum is likely to differ from the compile-time checksum due to gaps/unused bytes within the address range that the checksum is calculated over. Filling unused locations with a known value will correct this.

(964) unpaired nibble in -FILL value will be truncated *(Hexmate)*

The hexadecimal code given to the FILL option contained an incomplete byte. The incomplete byte (nibble) will be disregarded.

(965) -STRPACK option not yet implemented, option will be ignored *(Hexmate)*

This option currently is not available and will be ignored.

(966) no END record for HEX file "*" *(Hexmate)*

Intel HEX file did not contain a record of type END. The HEX file may be incomplete.

(967) unused function definition "*" (from line *) *(Parser)*

The indicated `static` function was never called in the module being compiled. Being static, the function cannot be called from other modules so this warning implies the function is never used. Either the function is redundant, or the code that was meant to call it was excluded from compilation or misspelled the name of the function.

(968) unterminated string *(Assembler)*

A string constant appears not to have a closing quote missing.

(969) end of string in format specifier *(Parser)*

The format specifier for the printf() style function is malformed.

(970) character not valid at this point in format specifier *(Parser)*

The printf() style format specifier has an illegal character.

(971) type modifiers not valid with this format *(Parser)*

Type modifiers may not be used with this format.

(972) only modifiers "h" and "l" valid with this format *(Parser)*

Only modifiers `h` (short) and `l` (long) are legal with this `printf` format specifier.

(973) only modifier "l" valid with this format *(Parser)*

The only modifier that is legal with this format is `l` (for long).

(974) type modifier already specified *(Parser)*

This type modifier has already be specified in this type.

(975) invalid format specifier or type modifier *(Parser)*

The format specifier or modifier in the printf-style string is illegal for this particular format.

(976) field width not valid at this point   *(Parser)*

A field width may not appear at this point in a printf() type format specifier.

(978) this identifier is already an enum tag   *(Parser)*

This identifier following a struct or union keyword is already the tag for an enumerated type, and thus should only follow the keyword enum, e.g.:

```
enum IN {ONE=1, TWO};
struct IN {              /* oops -- IN is already defined */
  int a, b;
};
```

(979) this identifier is already a struct tag   *(Parser)*

This identifier following a union or enum keyword is already the tag for a structure, and thus should only follow the keyword struct , e.g.:

```
struct IN {
  int a, b;
};
enum IN {ONE=1, TWO};  /* oops -- IN is already defined */
```

(980) this identifier is already a union tag   *(Parser)*

This identifier following a struct or enum keyword is already the tag for a union , and thus should only follow the keyword union , e.g.:

```
union IN {
  int a, b;
};
enum IN {ONE=1, TWO};  /* oops -- IN is already defined */
```

(981) pointer required   *(Parser)*

A pointer is required here, e.g.:

```
struct DATA data;
data->a = 9;      /* data is a structure,
                     not a pointer to a structure */
```

(982) unknown op "*" in nxtuse()   *(Assembler)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(983) storage class redeclared   *(Parser)*

A variable previously declared as being *static* , has now be redeclared as *extern*.

(984) type redeclared   *(Parser)*

The type of this function or object has been redeclared. This can occur because of two incompatible declarations, or because an implicit declaration is followed by an incompatible declaration, e.g.:

```
int a;
char a;  /* oops -- what is the correct type? */
```

(985) qualifiers redeclared   *(Parser)*

This function or variable has different qualifiers in different declarations.

(986) enum member redeclared *(Parser)*

A member of an enumeration is defined twice or more with differing values. Does the member appear twice in the same list or does the name of the member appear in more than one enum list?

(987) arguments redeclared *(Parser)*

The data types of the parameters passed to this function do not match its prototype.

(988) number of arguments redeclared *(Parser)*

The number of arguments in this function declaration does not agree with a previous declaration of the same function.

(989) module has code below file base of *h *(Linker)*

This module has code below the address given, but the `-C` option has been used to specify that a binary output file is to be created that is mapped to this address. This would mean code from this module would have to be placed before the beginning of the file! Check for missing psect directives in assembler files.

(990) modulus by zero in #if; zero result assumed *(Preprocessor)*

A modulus operation in a `#if` expression has a zero divisor. The result has been assumed to be zero, e.g.:

```
#define ZERO 0
#if FOO%ZERO    /* this will have an assumed result of 0 */
  #define INTERESTING
#endif
```

(991) integer expression required *(Parser)*

In an `enum` declaration, values may be assigned to the members, but the expression must evaluate to a constant of type `int` , e.g.:

```
enum {one = 1, two, about_three = 3.12};
  /* no non-int values allowed */
```

(992) can't find op *(Assembler)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(993) some command-line options are disabled *(Driver)*

The compiler is operating in demo mode. Some command-line options are disabled.

(994) some command-line options are disabled and compilation is delayed *(Driver)*

The compiler is operating in demo mode. Some command-line options are disabled, the compilation speed will be slower.

(995) some command-line options are disabled, code size is limited to 16kB, compilation is delayed *(Driver)*

The compiler is operating in demo mode. Some command-line options are disabled, the compilation speed will be slower, and the maximum allowed code size is limited to 16kB.

(1015) missing "*" specification in chipinfo file "*" at line *   *(Driver)*

This attribute was expected to appear at least once but was not defined for this chip.

(1016) missing argument* to "*" specification in chipinfo file "*" at line *
   (Driver)

This value of this attribute is blank in the chip configuration file.

(1017) extraneous argument* to "*" specification in chipinfo file "*" at line *   *(Driver)*

There are too many attributes for the listed specification in the chip configuration file.

(1018) illegal number of "*" specification* (* found; * expected) in chipinfo file "*" at line *   *(Driver)*

This attribute was expected to appear a certain number of times but it did not for this chip.

(1019) duplicate "*" specification in chipinfo file "*" at line *   *(Driver)*

This attribute can only be defined once but has been defined more than once for this chip.

(1020) unknown attribute "*" in chipinfo file "*" at line *   *(Driver)*

The chip configuration file contains an attribute that is not understood by this version of the compiler. Has the chip configuration file or the driver been replaced with an equivalent component from another version of this compiler?

(1021) syntax error reading "*" value in chipinfo file "*" at line *   *(Driver)*

The chip configuration file incorrectly defines the specified value for this device. If you are modifying this file yourself, take care and refer to the comments at the beginning of this file for a description on what type of values are expected here.

(1022) syntax error reading "*" range in chipinfo file "*" at line *   *(Driver)*

The chip configuration file incorrectly defines the specified range for this device. If you are modifying this file yourself, take care and refer to the comments at the beginning of this file for a description on what type of values are expected here.

(1024) syntax error in chipinfo file "*" at line *   *(Driver)*

The chip configuration file contains a syntax error at the line specified.

(1025) unknown architecture in chipinfo file "*" at line *   *(Driver)*

The attribute at the line indicated defines an architecture that is unknown to this compiler.

(1026) missing architecture in chipinfo file "*" at line *   *(Assembler)*

The chipinfo file has a processor section without an ARCH values. The architecture of the processor must be specified. Contact HI-TECH Support if the chipinfo file has not been modified.

(1027) activation was successful   *(Driver)*

The compiler was successfully activated.

(1028) activation was not successful - error code (*) *(Driver)*

The compiler did not activated successfully.

(1029) compiler not installed correctly - error code (*) *(Driver)*

This compiler has failed to find any activation information and cannot proceed to execute. The compiler may have been installed incorrectly or incompletely. The error code quoted can help diagnose the reason for this failure. You may be asked for this failure code if contacting HI-TECH Software for assistance with this problem.

(1030) Hexmate - Intel HEX editing utility (Build 1.%i) *(Hexmate)*

Indicating the version number of the `HEXMATE` being executed.

(1031) USAGE: * [input1.HEX] [input2.HEX]... [inputN.HEX] [options] *(Hexmate)*

The suggested usage of `HEXMATE`.

(1032) use –HELP=<option> for usage of these command line options *(Hexmate)*

More detailed information is available for a specific option by passing that option to the HELP option.

(1033) available command-line options: *(Hexmate)*

This is a simple heading that appears before the list of available options for this application.

(1034) type "*" for available options *(Hexmate)*

It looks like you need help. This advisory suggests how to get more information about the options available to this application or the usage of these options.

(1035) bad argument count (*) *(Parser)*

The number of arguments to a function is unreasonable. This is an internal compiler error. Contact HI-TECH Software technical support with details.

(1036) bad "*" optional header length (0x* expected) *(Cromwell)*

The length of the optional header in this COFF file was of an incorrect length.

(1037) short read on * *(Cromwell)*

When reading the type of data indicated in this message, it terminated before reaching its specified length.

(1038) string table length too short *(Cromwell)*

The specified length of the COFF string table is less than the minimum.

(1039) inconsistent symbol count *(Cromwell)*

The number of symbols in the symbol table has exceeded the number indicated in the COFF header.

(1040) bad checksum: record 0x*, checksum 0x* *(Cromwell)*

A record of the type specified failed to match its own checksum value.

(1041) short record *(Cromwell)*

While reading a file, one of the file's records ended short of its specified length.

(1042) unknown * record type 0x* *(Cromwell)*

The type indicator of this record did not match any valid types for this file format.

(1043) unknown optional header *(Cromwell)*

When reading this Microchip COFF file, the optional header within the file header was of an incorrect length.

(1044) end of file encountered *(Cromwell, Linker)*

The end of the file was found while more data was expected. Has this input file been truncated?

(1045) short read on block of * bytes *(Cromwell)*

A while reading a block of byte data from a UBROF record, the block ended before the expected length.

(1046) short string read *(Cromwell)*

A while reading a string from a UBROF record, the string ended before the specified length.

(1047) bad type byte for UBROF file *(Cromwell)*

This UBROF file did not begin with the correct record.

(1048) bad time/date stamp *(Cromwell)*

This UBROF file has a bad time/date stamp.

(1049) wrong CRC on 0x* bytes; should be * *(Cromwell)*

An end record has a mismatching CRC value in this UBROF file.

(1050) bad date in 0x52 record *(Cromwell)*

A debug record has a bad date component in this UBROF file.

(1051) bad date in 0x01 record *(Cromwell)*

A start of program record or segment record has a bad date component in this UBROF file.

(1052) unknown record type *(Cromwell)*

A record type could not be determined when reading this UBROF file.

(1053) additional RAM ranges larger than bank size *(Driver)*

A block of additional RAM being requested exceeds the size of a bank. Try breaking the block into multiple ranges that do not cross bank boundaries.

(1054) additional RAM range out of bounds *(Driver)*

The RAM memory range as defined through custom RAM configuration is out of range.

(1055) RAM range out of bounds (*) *(Driver)*

The RAM memory range as defined in the chip configuration file or through custom configuration is out of range.

(1056) unknown chip architecture *(Driver)*

The compiler is attempting to compile for a device of an architecture that is either unsupported or disabled.

(1057) fast double option only available on 17 series processors *(Driver)*

The fast double library cannot be selected for this device. These routines are only available for PIC17 devices.

(1058) assertion *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(1059) rewrite loop *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(1081) static initialization of persistent variable "*" *(Parser, Code Generator)*

A persistent variable has been assigned an initial value. This is somewhat contradictory as the initial value will be assigned to the variable during execution of the compiler's startup code, however the *persistent* qualifier requests that this variable shall be unchanged by the compiler's startup code.

(1082) size of initialized array element is zero *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(1088) function pointer "*" is used but never assigned a value *(Code Generator)*

A function call involving a function pointer was made, but the pointer was never assigned a target address, e.g.:

```
void (*fp)(int);
fp(23);     /* oops -- what function does fp point to? */
```

(1089) recursive function call to "*" *(Code Generator)*

A recursive call to the specified function has been found. The call may be direct or indirect (using function pointers) and may be either a function calling itself, or calling another function whose call graph includes the function under consideration.

(1090) variable "*" is not used *(Code Generator)*

This variable is declared but has not been used by the program. Consider removing it from the program.

(1091) main function "*" not defined *(Code Generator)*

The *main* function has not been defined. Every C program must have a function called *main*.

(1094) bad derived type *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(1095) bad call to typeSub() *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(1096) type should be unqualified *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(1097) unknown type string "*" *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(1098) conflicting declarations for variable "*" (*:*) *(Parser, Code Generator)*

Differing type information has been detected in the declarations for a variable, or between a declaration and the definition of a variable, e.g.:

```
extern long int test;
int test;    /* oops -- which is right? int or long int ? */
```

(1104) unqualified error *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(1118) bad string "*" in getexpr(J) *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(1119) bad string "*" in getexpr(LRN) *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(1121) expression error *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(1137) match() error: * *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(1157) W register must be W9 *(Assembler)*

The working register required here has to be W9, but an other working register was selected.

(1159) W register must be W11 *(Assembler)*

The working register required here has to be W11, but an other working register was selected.

(1178) the "*" option has been removed and has no effect *(Driver)*

This option no longer exists in this version of the compiler and has been ignored. Use the compiler's –*help* option or refer to the manual to find a replacement option.

(1179) interrupt level for function "*" may not exceed * *(Code Generator)*

The interrupt level for the function specified is too high. Each interrupt function is assigned a unique interrupt level. This level is considered when analyzing the call graph and reentrantly called functions. If using the `interrupt_level` pragma, check the value specified.

(1180) directory "*" does not exist *(Driver)*

The directory specified in the setup option does not exist. Create the directory and try again.

(1182) near variables must be global or static *(Code Generator)*

A variable qualified as *near* must also be qualified with *static* or made global. An auto variable cannot be qualified as *near*.

(1183) invalid version number *(Activation)*

During activation, no matching version number was found on the HI-TECH activation server database for the serial number specified.

(1184) activation limit reached *(Activation)*

The number of activations of the serial number specified has exceeded the maximum number allowed for the license.

(1185) invalid serial number *(Activation)*

During activation, no matching serial number was found on the HI-TECH activation server database.

(1186) licence has expired *(Driver)*

The time-limited license for this compiler has expired.

(1187) invalid activation request *(Driver)*

The compiler has not been correctly activated.

(1188) network error * *(Activation)*

The compiler activation software was unable to connect to the HI-TECH activation server via the network.

(1190) FAE license only - not for use in commercial applications *(Driver)*

Indicates that this compiler has been activated with an FAE license. This license does not permit the product to be used for the development of commercial applications.

(1191) licensed for educational use only *(Driver)*

Indicates that this compiler has been activated with an education license. The educational license is only available to educational facilities and does not permit the product to be used for the development of commercial applications.

(1192) licensed for evaluation purposes only *(Driver)*

Indicates that this compiler has been activated with an evaluation license.

(1193) this licence will expire on *   *(Driver)*

The compiler has been installed as a time-limited trial. This trial will end on the date specified.

(1195) invalid syntax for "*" option   *(Driver)*

A command line option that accepts additional parameters was given inappropriate data or insufficient data. For example an option may expect two parameters with both being integers. Passing a string as one of these parameters or supplying only one parameter could result in this error.

(1198) too many "*" specifications; * maximum   *(Hexmate)*

This option has been specified too many times. If possible, try performing these operations over several command lines.

(1199) compiler has not been activated   *(Driver)*

The trial period for this compiler has expired. The compiler is now inoperable until activated with a valid serial number. Contact HI-TECH Software to purchase this software and obtain a serial number.

(1200) Found %0*lXh at address *h   *(Hexmate)*

The code sequence specified in a -FIND option has been found at this address.

(1201) all FIND/REPLACE code specifications must be of equal width
                (Hexmate)

All find, replace and mask attributes in this option must be of the same byte width. Check the parameters supplied to this option. For example finding 1234h (2 bytes) masked with FFh (1 byte) will result in an error, but masking with 00FFh (2 bytes) will be okay.

(1202) unknown format requested in -FORMAT: *   *(Hexmate)*

An unknown or unsupported INHX format has been requested. Refer to documentation for supported INHX formats.

(1203) unpaired nibble in * value will be truncated   *(Hexmate)*

Data to this option was not entered as whole bytes. Perhaps the data was incomplete or a leading zero was omitted. For example the value Fh contains only four bits of significant data and is not a whole byte. The value 0Fh contains eight bits of significant data and is a whole byte.

(1204) * value must be between 1 and * bytes long   *(Hexmate)*

An illegal length of data was given to this option. The value provided to this option exceeds the maximum or minimum bounds required by this option.

**(1205) using the configuration file *; you may override this with the
                environment variable HTC_XML**   *(Driver)*

This is the compiler configuration file selected during compiler setup. This can be changed via the HTC_XML environment variable. This file is used to determine where the compiler has been installed.

(1207) some of the command line options you are using are now obsolete *(Driver)*

Some of the command line options passed to the driver have now been discontinued in this version of the compiler, however during a grace period these old options will still be processed by the driver.

(1208) use –help option or refer to the user manual for option details *(Driver)*

An obsolete option was detected. Use –help or refer to the manual to find a replacement option that will not result in this advisory message.

(1209) An old MPLAB IDE tool suite plug-in was detected. *(Driver)*

The options passed to the driver resemble those that the Microchip MPLAB IDE would pass to a previous version of this compiler. Some of these options are now obsolete, however they were still interpreted. It is recommended that you install an updated HI-TECH options plug-in for the MPLAB IDE.

(1210) Visit the HI-TECH Software website (www.htsoft.com) for a possible update *(Driver)*

Visit our website to see if an update is available to address the issue(s) listed in the previous compiler message. Please refer to the on-line self-help facilities such as the *Frequently asked Questions* or search the *On-line forums*. In the event of no details being found here, contact HI-TECH Software for further information.

(1212) Found * (%0*lXh) at address *h *(Hexmate)*

The code sequence specified in a -FIND option has been found at this address.

(1213) duplicate ARCH for * in chipinfo file at line * *(Assembler, Driver)*

The chipinfo file has a processor section with multiple ARCH values. Only one ARCH value is allowed. If you have not manually edited the chip info file, contact HI-TECH Support with details.

(1218) can't create cross reference file * *(Assembler)*

The assembler attempted to create a cross reference file, but it could not be created. Check that the file's pathname is correct.

(1228) unable to locate installation directory *(Driver)*

The compiler cannot determine the directory where it has been installed.

(1230) dereferencing uninitialized pointer "*" *(Code Generator)*

A pointer that has not yet been assigned a value has been dereferenced. This can result in erroneous behavior at runtime.

(1235) unknown keyword * *(Driver)*

The token contained in the USB descriptor file was not recognized.

(1236) invalid argument to *: * *(Driver)*

An option that can take additional parameters was given an invalid parameter value. Check the usage of the option or the syntax or range of the expected parameter.

(1237) endpoint 0 is pre-defined *(Driver)*

An attempt has been made to define endpoint 0 in a USB file. This channel c

(1238) FNALIGN failure on * *(Linker)*

Two functions have their auto/parameter blocks aligned using the `FNALIGN` directive, but one function calls the other, which implies that must not be aligned. This will occur if a function pointer is assigned the address of each function, but one function calls the other. For example:

```
int one(int a) { return a; }
int two(int a) { return two(a)+2; }  /* ! */
int (*ip)(int);
ip = one;
ip(23);
ip = two;       /* ip references one and two; two calls one */
ip(67);
```

(1239) pointer * has no valid targets *(Code Generator)*

A function call involving a function pointer was made, but the pointer was never assigned a target address, e.g.:

```
void (*fp)(int);
fp(23);     /* oops -- what function does fp point to? */
```

(1240) unknown checksum algorithm type (%i) *(Driver)*

The error file specified after the `-Efile` or `-E+file` options could not be opened. Check to ensure that the file or directory is valid and that has read only access.

(1241) bad start address in * *(Driver)*

The start of range address for the `--CHECKSUM` option could not be read. This value must be a hexadecimal number.

(1242) bad end address in * *(Driver)*

The end of range address for the `--CHECKSUM` option could not be read. This value must be a hexadecimal number.

(1243) bad destination address in * *(Driver)*

The destination address for the `--CHECKSUM` option could not be read. This value must be a hexadecimal number.

(1245) value greater than zero required for * *(Hexmate)*

The *align* operand to the `HEXMATE -FIND` option must be positive.

(1246) no RAM defined for variable placement *(Code Generator)*

No memory has been specified to cover the banked RAM memory.

(1247) no access RAM defined for variable placement *(Code Generator)*

No memory has been specified to cover the access bank memory.

(1248) symbol (*) encountered with undefined type size *(Code Generator)*

The code generator was asked to position a variable, but the size of the variable is not known. This is an internal compiler error. Contact HI-TECH Software technical support with details.

(1250) could not find space (* byte*) for variable * *(Code Generator)*

The code generator could not find space in the banked RAM for the variable specified.

(1253) could not find space (* byte*) for auto/param block *(Code Generator)*

The code generator could not find space in RAM for the psect that holds `auto` and parameter variables.

(1254) could not find space (* byte*) for data block *(Code Generator)*

The code generator could not find space in RAM for the data psect that holds initialized variables.

(1255) conflicting paths for output directory *(Driver)*

The compiler has been given contradictory paths for the output directory via any of the `-O` or `--OUTDIR` options, e.g.

```
--outdir=../../   -o../main.HEX
```

(1256) undefined symbol "*" treated as HEX constant *(Assembler)*

A token which could either be interpreted as a symbol or a hexadecimal value does not match any previously defined symbol and so will be interpreted as the latter. Use a leading *zero* to avoid the ambiguity, or use an alternate radix specifier such as `0x`. For example:

```
MOV  a, F7h  ; is this the symbol F7h, or the HEX number 0xF7?
```

(1257) local variable "*" is used but never given a value *(Code Generator)*

An `auto` variable has been defined and used in an expression, but it has not been assigned a value in the C code before its first use. Auto variables are not cleared on startup and their initial value is undefined. For example:

```
void main(void) {
  double src, out;
  out = sin(src);   /* oops -- what value was in src? */
```

(1258) possible stack overflow when calling function "*" *(Code Generator)*

The call tree analysis by the code generator indicates that the hardware stack may overflow. This should be treated as a guide only. Interrupts, the assembler optimizer and the program structure may affect the stack usage. The stack usage is based on the C program and does not include any call tree derived from assembly code.

(1259) can't optimize for both speed and space *(Driver)*

The driver has been given contradictory options of compile for speed and compile for space, e.g.

```
--opt=speed,space
```

(1260) macro "*" redefined   *(Assembler)*

More than one definition for a macro with the same name has been encountered, e.g.

```
MACRO fin
  ret
ENDM
MACRO fin   ; oops -- was this meant to be a different macro?
  reti
ENDM
```

(1261) string constant required   *(Assembler)*

A string argument is required with the DS or DSU directive, e.g.

```
DS ONE     ; oops -- did you mean DS "ONE"?
```

(1264) unsafe pointer conversion   *(Code Generator)*

A pointer to one kind of structure has been converted to another kind of structure and the structures do not have a similar definition, e.g.

```
struct ONE {
  unsigned a;
  long b;           /* ! */
} one;
struct TWO {
  unsigned a;
  unsigned b;       /* ! */
} two;
struct ONE * oneptr;
oneptr = & two;     /* oops --
                    was ONE meant to be same struct as TWO? */
```

(1267) fixup overflow referencing * into * bytes at 0x*   *(Linker)*

See the following error message (1268) for more information.

(1268) fixup overflow storing 0x* in * bytes at *   *(Linker)*

Fixup is the process conducted by the linker of replacing symbolic references to variables etc, in an assembler instruction with an absolute value. This takes place after positioning the psects (program sections or blocks) into the available memory on the target device. Fixup overflow is when the value determined for a symbol is too large to fit within the allocated space within the assembler instruction. For example, if an assembler instruction has an 8-bit field to hold an address and the linker determines that the symbol that has been used to represent this address has the value 0x110, then clearly this value cannot be inserted into the instruction.

(0) delete what ? *(Libr)*

The librarian requires one or more modules to be listed for deletion when using the d key, e.g.:

```
libr d c:\ht-pic\lib\pic704-c.lib
```

does not indicate which modules to delete. try something like:

```
libr d c:\ht-pic\lib\pic704-c.lib wdiv.obj
```

(0) incomplete ident record *(Libr)*

The IDENT record in the object file was incomplete. Contact HI-TECH Support with details.

(0) incomplete symbol record *(Libr)*

The SYM record in the object file was incomplete. Contact HI-TECH Support with details.

(0) library file names should have.lib extension: * *(Libr)*

Use the .lib extension when specifying a library filename.

(0) module * defines no symbols *(Libr)*

No symbols were found in the module's object file. This may be what was intended, or it may mean that part of the code was inadvertently removed or commented.

(0) replace what ? *(Libr)*

The librarian requires one or more modules to be listed for replacement when using the r key, e.g.:

```
libr r lcd.lib
```

This command needs the name of a module (.obj file) after the library name.

NOTES:

Index

Index

Index

WORLDWIDE SALES AND SERVICE

AMERICAS

Corporate Office
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200
Fax: 480-792-7277
Technical Support:
http://support.microchip.com
Web Address:
www.microchip.com

Atlanta
Duluth, GA
Tel: 678-957-9614
Fax: 678-957-1455

Boston
Westborough, MA
Tel: 774-760-0087
Fax: 774-760-0088

Chicago
Itasca, IL
Tel: 630-285-0071
Fax: 630-285-0075

Cleveland
Independence, OH
Tel: 216-447-0464
Fax: 216-447-0643

Dallas
Addison, TX
Tel: 972-818-7423
Fax: 972-818-2924

Detroit
Farmington Hills, MI
Tel: 248-538-2250
Fax: 248-538-2260

Kokomo
Kokomo, IN
Tel: 765-864-8360
Fax: 765-864-8387

Los Angeles
Mission Viejo, CA
Tel: 949-462-9523
Fax: 949-462-9608

Santa Clara
Santa Clara, CA
Tel: 408-961-6444
Fax: 408-961-6445

Toronto
Mississauga, Ontario,
Canada
Tel: 905-673-0699
Fax: 905-673-6509

ASIA/PACIFIC

Asia Pacific Office
Suites 3707-14, 37th Floor
Tower 6, The Gateway
Harbour City, Kowloon
Hong Kong
Tel: 852-2401-1200
Fax: 852-2401-3431

Australia - Sydney
Tel: 61-2-9868-6733
Fax: 61-2-9868-6755

China - Beijing
Tel: 86-10-8528-2100
Fax: 86-10-8528-2104

China - Chengdu
Tel: 86-28-8665-5511
Fax: 86-28-8665-7889

China - Hong Kong SAR
Tel: 852-2401-1200
Fax: 852-2401-3431

China - Nanjing
Tel: 86-25-8473-2460
Fax: 86-25-8473-2470

China - Qingdao
Tel: 86-532-8502-7355
Fax: 86-532-8502-7205

China - Shanghai
Tel: 86-21-5407-5533
Fax: 86-21-5407-5066

China - Shenyang
Tel: 86-24-2334-2829
Fax: 86-24-2334-2393

China - Shenzhen
Tel: 86-755-8203-2660
Fax: 86-755-8203-1760

China - Wuhan
Tel: 86-27-5980-5300
Fax: 86-27-5980-5118

China - Xiamen
Tel: 86-592-2388138
Fax: 86-592-2388130

China - Xian
Tel: 86-29-8833-7252
Fax: 86-29-8833-7256

China - Zhuhai
Tel: 86-756-3210040
Fax: 86-756-3210049

ASIA/PACIFIC

India - Bangalore
Tel: 91-80-3090-4444
Fax: 91-80-3090-4080

India - New Delhi
Tel: 91-11-4160-8631
Fax: 91-11-4160-8632

India - Pune
Tel: 91-20-2566-1512
Fax: 91-20-2566-1513

Japan - Yokohama
Tel: 81-45-471- 6166
Fax: 81-45-471-6122

Korea - Daegu
Tel: 82-53-744-4301
Fax: 82-53-744-4302

Korea - Seoul
Tel: 82-2-554-7200
Fax: 82-2-558-5932 or
82-2-558-5934

Malaysia - Kuala Lumpur
Tel: 60-3-6201-9857
Fax: 60-3-6201-9859

Malaysia - Penang
Tel: 60-4-227-8870
Fax: 60-4-227-4068

Philippines - Manila
Tel: 63-2-634-9065
Fax: 63-2-634-9069

Singapore
Tel: 65-6334-8870
Fax: 65-6334-8850

Taiwan - Hsin Chu
Tel: 886-3-6578-300
Fax: 886-3-6578-370

Taiwan - Kaohsiung
Tel: 886-7-536-4818
Fax: 886-7-536-4803

Taiwan - Taipei
Tel: 886-2-2500-6610
Fax: 886-2-2508-0102

Thailand - Bangkok
Tel: 66-2-694-1351
Fax: 66-2-694-1350

EUROPE

Austria - Wels
Tel: 43-7242-2244-39
Fax: 43-7242-2244-393

Denmark - Copenhagen
Tel: 45-4450-2828
Fax: 45-4485-2829

France - Paris
Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

Germany - Munich
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

Italy - Milan
Tel: 39-0331-742611
Fax: 39-0331-466781

Netherlands - Drunen
Tel: 31-416-690399
Fax: 31-416-690340

Spain - Madrid
Tel: 34-91-708-08-90
Fax: 34-91-708-08-91

UK - Wokingham
Tel: 44-118-921-5869
Fax: 44-118-921-5820

03/26/09